AFRL-RI-RS-TR-2016-277

DETECTION OF MALWARE COLLUSION WITH STATIC DEPENDENCE ANALYSIS ON INTER-APP COMMUNICATION

VIRGINIA TECH

DECEMBER 2016

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2016-277 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
MARK K. WILLIAMS
Work Unit Manager

/ S /
WARREN H. DEBANY, JR.
Technical Advisor, Information
Exploitation and Operations Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
DECEMBER 2016	FINAL TECHNICAL REPORT	MAR 2015 – SEP 2016

4. TITLE AND SUBTITLE

DETECTION OF MALWARE COLLUSION WITH STATIC DEPENDENCE ANALYSIS ON INTER-APP COMMUNICATION

5a. CONTRACT NUMBER
FA8750-15-2-0076

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Danfeng Yao, Barbara Ryder

5d. PROJECT NUMBER
APAC

5e. TASK NUMBER
VT

5f. WORK UNIT NUMBER
EC

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Virginia Polytechnic Institute & State University
Virginia Tech
300 Turner St NW,Suite 4200
Blacksburg, VA 24061-0001

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RIGB
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2016-277

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Inter-Component Communication (ICC) enables useful interactions between mobile apps. It has been long believed that inter-app ICCs can be abused by malware writers to launch collusion attacks. In this project, we designed and developed two techniques for analyzing the security threats associated with inter-app ICC communications and conducted two large-scale experiments (MapReduce one with 11,996 apps and MySQL database one with over 110K apps). Our contributions are two-fold, methodology development and empirical analysis. We invented two complementary methods for efficiently screening Android app pairs against data leak and privilege escalation threats that are due to intentional collusions or vulnerable apps being exploited.

15. SUBJECT TERMS

Malware Collusion; Inter-App Communication; Static Dependence Analysis

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**MARK WILLIAMS**
U	U	U	UU	51	19b. TELEPHONE NUMBER (*Include area code*) **(315) 330-4560**

Table of Contents

Section **Page**

List of Figures

Figure **Page**

List of Tables

Table **Page**

1 SUMMARY

Inter-Component Communication (ICC) enables useful interactions between mobile apps. It has been long believed that inter-app ICCs can be abused by malware writers to launch collusion attacks. Because of the evolution nature of malware development, this next-generation collusive malware, where each app may appear simple and benign in order to evade the existing single-app screening, is indeed conceivable. However, because of the complexity of performing pairwise program analysis on apps, the scale of existing inter-app analyses is too small (e.g., up to several hundreds) to provide sufficient security insights. Existing single-app ICC analysis is inadequate in solving the pairwise app security problems.

In this project, we designed and developed two techniques for analyzing the security threats associated with inter-app ICC communications and conducted two large-scale experiments (MapReduce one with 11,996 apps and MySQL database one with over 110K apps). Our contributions are two-fold, methodology development and empirical analysis. We invented two complementary methods for efficiently screening Android app pairs against data leak and privilege escalation threats that are due to intentional collusions or vulnerable apps being exploited. Our prototypes produce a trove of pairwise data-flow information that provides valuable new security insights, as well as related algorithms and software.

First, we present MR-Droid, a MapReduce-based computing framework for parallel inter-app ICC analysis in Android. MR-Droid extracts data-flow features between multiple communicating apps and the target apps to build a large-scale ICC graph. We leverage the ICC graph to provide contexts for inter-app communications to produce precise alerts and prioritize risk assessments. This process requires large app-pair data, which is enabled by our MapReduce-based program analysis. Extensive experiments on 11,996 apps from 24 app categories demonstrate the effectiveness of our risk prioritization scheme. Our analyses with MR-Droid reveal real-world hijacking attacks and colluded app pairs. We provide practical recommendations for reducing inter-app communication risks. MR-Droid analyzes ICC interfaces, but does not perform internal data-flow analysis within each app.

Second, we present DIALDroid, a MySQL-based inter-app analysis framework that performs in-depth data-flow analysis beyond the ICC interfaces. DIALDroid improves MR-Droid and provides much more fine-grained and informed data-flow characterization of app pairs. We also developed a new ICC resolution tool in DIALDroid for matching intent fields. We report our findings in large-scale collusion and vulnerable app detection. Our analysis examines the inter-app ICCs data flows among more than 110,150 real-world apps. DIALDroid's design aims to balance the accuracy of static ICC resolution and data-flow analysis and the performance.

Our large-scale analysis with DIALDroid provides real-world evidences on various types of inter-app ICC abuse, including implicit-intent based external data leaks (app A retrieves sensitive data, passes to app B, who leaks it via disk or network output) and privilege escalations. Our data analytics follow a fine-grained categorization of threats according to data-flow features. We found no explicit-intent based collusion or privilege escalation cases, but many implicit intent ones. Besides the empirical findings, we make several technical contributions, including a new open-source ICC resolution tool with improved accuracy than the state-of-the-art and a database for inter-app ICC research.

2 INTRODUCTION

2.1 Overview

Inter-Component Communication (ICC) is an important mechanism for app-to-app communications on Android. It links the components of different apps via messaging objects (or Intents). While ICC contributes greatly to the development of rich third-party applications, this communication model has become a predominant security attack surface. In the context of inter-app communication scenarios, individual apps often suffer from risky vulnerabilities such as Intent hijacking and spoofing, resulting in leaking sensitive user data [7]. More importantly, ICC allows two or more malicious apps to collude on stealthy attacks that none of them could accomplish alone [4, 25]. According to a recent report from McAfee Labs [1], app collusions are increasingly prevalent on mobile platforms.

To assess ICC vulnerabilities, various analytics methods have been proposed, ranging from intra-app taint flow discovery [2] to inter-app communication vulnerability detection [7, 30]. Most of them perform analysis for one individual app at a time, ignoring its feasible communication context with other apps. Single-app analyses provide conservative risk estimations, producing a high number of (false) alarms [7, 30]. For example, ComDroid [7] reported that over 97% of top popular apps are vulnerable to inter-app attacks. Manually investigating all the alerts would be a daunting task for security analysts. Other approaches either tune for fewer false alerts only for specific inter-app attacks (e.g., privilege escalation attacks) [4] or resort to heavyweight (yet not scalable) analysis to achieve higher precision [2, 12, 38].

A more effective approach for characterizing collusive or vulnerable ICCs is inter-app analyses that consider ICCs across two or more apps. This allows researchers to gain empirical contexts on the actual communications between apps and produce more relevant alerts, e.g., identifying privacy leak [21] and permission escalations [3, 34]. However, existing solutions are largely limited in scale due to the high complexity of pair-wise components analyses ($O(N^2)$ for N apps). They were either applied to a much smaller set of apps (a few hundred only, versus a few thousand in single-app analyses), or small set of inter-app links.

In this project, we successfully developed two (2) complementary techniques, MR-Droid and DIALDroid, for large-scale inter-app ICC analysis against the threats posed by collusion and vulnerable apps. MR-Droid supports parallel computing enabled by MapReduce framework. DIALDroid uses relational database for compact storage and fast ICC linking. MR-Droid analyzes the ICC interfaces, and DIALDroid also performs in-depth data-flow analysis within each app. We used MR-Droid to analyze 12K apps, and used DIALDroid to analyze 110K apps. Next, we give an overview of both methods.

2.2 MR-Droid Overview

First, MR-Droid is a MapReduce-based parallel analytics system for accurate and scalable ICC risk detection. Our goal is to empirically evaluate ICC risks based on an app's inter-connections with other real-world apps and efficiently identify high-risk pairs. Our intuition is that an ICC pair is high-risk not only because one of the apps is vulnerable, but more importantly it potentially communicates with other apps. To achieve this goal, we construct a large-scale ICC graph, where each node is an app

component and the edge represents the corresponding inter-app ICCs[1]. To gauge the risk level of the ICC pairs (edge weight), we extract various features based on app flow analyses that indicate vulnerabilities. For instance, we examine whether the ICC pair is used to pass sensitive data, or escalate permissions for another app. With the ICC graph, we then rank the risk level of a given app by aggregating all its ICC edges with other apps.

Our system allows app market administrators to accurately pinpoint market-wise high-risk apps and prioritize risk migration. Individual app developers can also leverage our results to assess their own apps. In addition, the ICC graph provides rich contexts on how an app vulnerability is exploited, and by (or with) which external apps. For the purpose of this work, we customize our features to capture three major ICC risks: app collusion (malicious apps working together on stealth attacks), intent hijacking and intent spoofing (vulnerabilities that allow unauthorized app to eavesdrop or manipulate inter-app communications).

To scale up the system, we implement MR-Droid with a set of new MapReduce [9] algorithms atop the Hadoop framework. We carefully decouple the dependencies during the ICC graph construction to reduce complexity. Instead of jointly analyzing all app pairs ($O(N^2)$), we independently exact ICC sources and sinks from each app, and only jointly analyze those with matched source-sink pairs. Due to the relatively sparse inter-connections among apps, we can achieve near-linear complexity. The high-level parallelization from MapReduce allows us to analyze millions of app pairs within hours using commodity servers.

We evaluate our systems on a large set of 11,996 Android apps collected from 24 major Google Play categories (13 million ICC pairs). We then use substantial manual analysis to confirm the results. Evaluation confirms the effectiveness of our approach in reducing false, excessive alerts. For apps labeled as high-risk, we obtain a 100% true positive rate in detecting collusion, broadcast injection, activity- and service-launch based intent spoofing, and a 90% true positive rate for activity hijacking and broadcast theft detection. For app pairs labeled as medium- or low-risk, manual analysis show their actual risks are substantially lower, indicating the effectiveness of risk prioritization.

Our system is highly scalable. With 15 commodity servers, the entire process took less than 25 hours for an average of only 0.0012 seconds per app pair. More importantly, our runtime experiments show the computation time grows near-linearly with respect to the number of apps.

Our empirical analysis also reveals valuable new insights to app collusion and hijacking attacks. We find previously unknown real-world colluding apps that leak user data (e.g., passwords) and escalate each other's permission (e.g., location or network access).

In addition, we find a more stealth way of collusion using implicit intents. Instead of "explicitly" communicating with each other (easy to detect), the colluding apps using high customized (rare) actions to mark their implicit intent to achieve the same effect of explicit intent. This type of collusion cannot be detected by analyzing each app independently. Finally, we find third-party libraries and the automatically-generated apps have contributed greatly to today's hijacking vulnerabilities.

Our work makes four key contributions.

- We propose an empirical analytics method to identify risks within inter-app communications (or ICC). By constructing a large ICC graph, we assess an app's ICC risks based on its empirical communications with other apps. Using a risk ranking scheme, we accurately identify high-risk apps.
- We design and implement a highly scalable MapReduce framework for our inter-app ICC analyses. With carefully designed MapReduce cycles, we avoid full pair-wise ICC analyses ($O(N^2)$) and achieve near-linear complexity.

[1] We use ICC graph to represent inter-app communications throughout the report.

- We evaluate our system on 11,996 top Android apps under 24 Google Play categories (13 million ICC pairs). Our evaluation confirms the effectiveness of our approach in reducing false alerts (90%-100% true positive rate) and its high scalability.
- Our empirical analysis reveals new types of app collusion and hijacking risks (e.g., transferring user's sensitive information including password[2] and leveraging rarely used implicit intents[3]). Based on our results, we provide practical recommendations for app developers to reduce ICC vulnerabilities.

2.3 DIALDroid Overview

We developed a scalable and accurate tool DIALDroid for inter-app ICC security analysis. We used DIALDroid to perform the first systematic large-scale security analysis on inter-app data-flows among 100,206 real-world apps from Google Play. Some of our scalability analysis also includes 9,944 malware apps from Virus Share. DIALDroid completes such a large-scale analysis within a reasonable time frame (6,340 total hours of program analysis and 82 minutes of ICC linking and detection). We are able to balance the accuracy and scalability. Our key designs include the adaptive and pragmatic flow analysis configuration, fast query in an optimized relational database, and precise ICC resolution. Our work with DIALDroid provides new empirical evidences on app collusion and privilege escalation. We summarize our contributions as follows.

- We develop an Android security tool, DIALDroid, for analyzing ICC-based sensitive inter-app data flows. Our design DIALDroid leverages relational database for scalable matching of ICC entry and exit points, and fast analysis-query response. DIALDroid outperforms state-of-art solutions in terms of both accuracy and runtime performance on benchmark apps. E.g., DIALDroid substantially outperforms IccTA+ApkCombiner and COVERT with precision 91.7% and recall 95.6% on benchmark evaluation. (IccTA [21] is designed for intra-app ICCs in a single app. It needs to use ApkCombiner [20] to combine app-pairs for inter-app ICC analysis.) For runtime, DIALDroid is orders of magnitude faster with less crashes. In addition, DIALDroid's ICC extractor is more accurate than the state-of-the-art solution IC3 [29], with 28% more identified intents and 33% less failed cases.
- We describe an approach for categorizing and characterizing the security threats of inter-app data flows. Our categorization is based on the sensitivity of the ICC exits (in the source app), the functionality of the ICC entries (in the receiver app), the change in privileges of the receiver app, as well as the type of the intent. We obtain 6 threat categories for pairwise analysis. We use DIALDroid to analyze the inter-app ICCs among 100,206 apps from the Google Play Market according to multiple threat categories.
- Our datasets and tools can potentially benefit the broader Android community. We have released our ICC extractor tool IC3-DIALDroid in GitHub (https://github.com/dialdroid-android/ic3-dialdroid). We are in the process of releasing our full database to the public. Our database contains an extremely rich set of data-flow attributes of 100,206 apps from the Google Play Market and 9,944 apps from the Virus Share database. These attributes are extracted from running FlowDroid static program analysis and are organized in a number of relational tables. We envision that the database can be used by both the security and data mining communities to

[2] *uda.onsiteplanroom and uda.projectlogging*

[3] *org.geometerplus.fbreader.plugin.local_opds_sanner* and *com.vng.android.zingbrowser.labanbo-okreader*

tackle open research questions in Android. In addition, we have released a benchmark suite for inter-app ICC security analysis, DIALDroid-Bench (https://github.com/dialdroid-android/dialdroid-bench/), which contains 30 real-world apps from Google play. To our knowledge, this set of apps is the first such benchmark using real-world apps, as opposed to proof-of-concept apps.

2.4 Related Work

Inter-app Attacks. Attacks with multiple apps involved have been proposed recently. Chin et al. [7] analyzed the inter-app vulnerabilities in Android. They pointed out that the message passing system involves various inter-app attacks including broadcast theft, activity hijacking, etc. Davi et al. [8] conducted a permission escalation attack by invoking higher-privileged apps, which do not sufficiently protect their interfaces, from a non-privileged app. Ren et al. [33] demonstrated intent hijacking can lead to UI spoofing, denial-of-service and user monitoring attacks. Soundcomber [35] is a malicious app that transmits sensitive data to the Internet through an overt/covert channel to a second app. Android browsers also become the target of inter-app attacks via intent scheme URLs [37] and Cross-Application Scripting [13]. Inter-app attacks have become a serious security problem on smartphones.

Defense and Detection Techniques. A significant number of solutions have been proposed to perform single-app analysis and detect ICC vulnerabilities. DroidSafe [12] and AppIntent [40] analyzes sensitive data flow within an app and detects whether the sensitive information that flows out of the phone is user intended or not. ComDroid [7] and Epicc [30] identify Intents specifications on Android apps and detect ICC vulnerabilities. CHEX [23] discovers entry points and detects hijacking-enabling data flows in the app for component hijacking (Intent spoofing). Mutchler et al. [27] studied the security of mobile web apps. In particular, they detected inter-app URLs leakage in the mobile web apps. Zhang and Yin [42] proposed to automatically patch component hijacking vulnerabilities. Mulliner et al. [26] presented a scalable approach to apply third-party security patches against privilege escalation, capability leaks etc. PermissionFlow [34] detects inter-application permission leaks using taint analysis. On-device policies [6, 17, 31] were also designed to detect inter-app vulnerabilities and collusion attacks.

Inter-app analysis has also been proposed in the literature. One may combine multiple apps into one and perform single-app analysis on the combined one [20, 21]. These approaches include flow-analysis and gain more detailed flow information from it. However, they do not scale and would be extremely expensive if applied to a large set of apps. In addition, in straightforward implementations, expensive program analysis is performed repetitively, which is redundant. In comparison, we perform data-flow analysis of an app only once, independent of how many its neighbors are.

COVERT [3] performs static analysis on multiple apps for detecting permission leakage. It extracts a model from each app. A formal analyzer is used to verify the safety of a given combination of apps. FUSE [32] statistically analyzes the binary of each individual app and merges the results to generate a multi-app information flow graph. The graph can be used for security analysis. Klieber et al. [18] performed static taint analysis on a set of apps with FlowDroid [2] and Epicc [30]. Jing et al. [16] proposed intent space analysis and a policy checking framework to identify the links among a small scale apps. The existing approaches mostly focus on the precision and analysis of in-depth flow information between apps. However, they are not designed to analyze large-scale apps. It is unclear how well they scale with real world apps. In addition, some solutions (such as Epicc and IC3) are for identifying ICCs, but do not have in-depth security classification as our work.

PRIMO [28] reported ICC analysis of a large pool of apps. Its analysis is on the likelihood of communications between two apps, not specifically on security. It does not provide any security classification. Nevertheless, PRIMO could potentially be used by us as a pre-processing step.

For related dynamic analysis, IntentFuzzer [39] detects capability leaks by dynamically sending Intents to the exposed interfaces. INTENTDROID [14] performs dynamic testing on Android apps and detects the vulnerabilities caused by unsafe handling of incoming ICC message. SCanDroid [11] checks whether the data flows through the apps are consistent with the security specifications from manifests. TaintDroid [10] tracks information flows with dynamic analysis and performs real-time privacy monitoring on Android. Similarly, FindDroid [44] associates each permission request with its application context thus protecting sensitive resources from unauthorized use. XManDroid [5] was proposed to prevent privilege escalation attacks and collusion attacks by analyzing the communications among apps and ensuring that the communications comply to a desired system policy. These dynamic analyses complement our static-analysis solution. However, their feasibility under a large number of app pairs is limited.

Generic Flow Analysis. FlowDroid [2] is a popular static taint analysis tool for Android apps. The user can define the sources and sinks within one app and discover whether there are connections from the sources to the sinks. Amandroid [38] tracks the inter-component data and control flow of an app. The analysis can be leveraged to perform various types of security analysis. IC3 [29] focuses on inter-component communications and inferring ICC specifications. Our work leverages IC3 [29] to extract the specifications of Intents. Although it is the most time-consuming step in our prototype, it is much more memory and computation efficient than other static analysis tools. In addition, our approach scales well and has the potentials for market-scale security analysis.

MapReduce for Large Scale Analysis. MapReduce framework has been used in various areas such as data mining [15], data management [19], and network/internet security [43, 22]. Our approach leverages the MapReduce framework to analyze large-scale apps for security analysis. To the best of our knowledge, it is the first distributed-computing solution tailored for addressing Android security problems.

3 METHODS

3.1 MR-Droid Computational Goal

The computational goal in MR-Droid is two-fold.

1. Build a complete inter-app ICC graph and identify all communication app pairs for a set of apps to provide the communication context (i.e., the neighbor set) for each one.
2. Further perform neighbor-aware inter-app security analysis on top of the ICC graph and rank the apps and app pairs with respect to their risk levels. The definition for an ICC graph is given in Definition 1.

DEFINITION 1, Inter-app ICC graph is a directed bipartite graph $G = (V_S, V_D, E_{V_S \to V_D})$*, where each node* $v \in V_S \cup V_D$ *represents the specifications of an entry or exit point of external ICC of an app, and each edge* $e \in E_{V_S \to V_D}$ *represents the Intent-based communication between one app's ICC exit point* $v_s \in V_S$ *to another app's ICC entry point* $v_d \in V_D$*. We refer to a node in the set* V_S*in G as an ICC source or simply source. We refer to a node in the set* V_D *as an ICC sink or simply sink.*

Figure 1 illustrates these definitions. In Table 1, we list some of the attribute details for sources and sinks. Sources represent the outbound Intents. Sinks represent Intent filter and/or public components. Given apps, the construction of ICC graph requires i) identifying communication nodes (Intent, Intent filter, or component) and ii) identifying edges (properties of ICCs). Identifying edges requires attributes matching and testing between sources and sinks. Both nodes and edges information should be extracted from apps.

Table 1: Nodes in ICC Graph in MR-Droid. Type is Either "Activity", "Service" or "Broadcast". pointType is either "Explicit", "Implicit" or "sharedUserId".

Node	Attributes
Source	*ID, type, destPackage, destComponent, action, categoryList, mimeType, uri, extraDataList, Permission, sharedUserId, pointType*
Sink	*ID, type, componentName, actionList, categoryList, permission, priority, mimeTypeList, schemeList, hostList, portList, pathList, pointType*

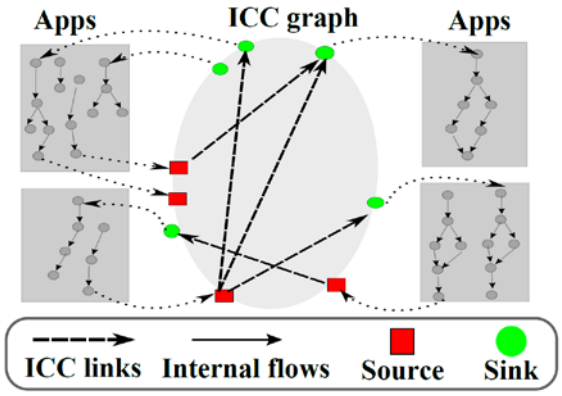


Figure 1: Illustration of Data Flows, Inter-app Intent Matching, and ICC Graph in MR-Droid. Our MapReduce Algorithms Compute the Complete ICC Graph Information For a Set of Apps. We Exclude Internal ICCs.

3.2 MR-Droid Workflow

As shown in Figure 2, our workflow involves a preprocessing step 1. Identify ICC Nodes, two MapReduce jobs referred to as 2. Identify ICC Edges, 3. Group ICCs Per App Pair, and a 4. Risk Analysis operation. Each MapReduce job consists of a Map and a Reduce algorithm.

1. In Identify ICC Nodes, we extract the attributes of sources and sinks from apps. Sources are extracted from the attributes in outbound Intents. Sinks are extracted from the exported components in the manifest or from dynamic receivers created in the code.
2. Identify ICC Edges is the first MapReduce job which identifies edges between communicating sources and sinks. The MapReduce job transforms the source and sinks into <key, value> pairs, which enable parallel edge finding.

3. Group ICCs Per App Pair is the second MapReduce job which performs the data test and permission checking, and identifies and groups edges belonging to the same pair.
4. Risk Analysis utilizes the ICC analysis results from previous phases to compute key ICC link features, and then uses the features to rank security risks for each app (for hijacking and spoofing attacks) and app pair (for collusion attacks).

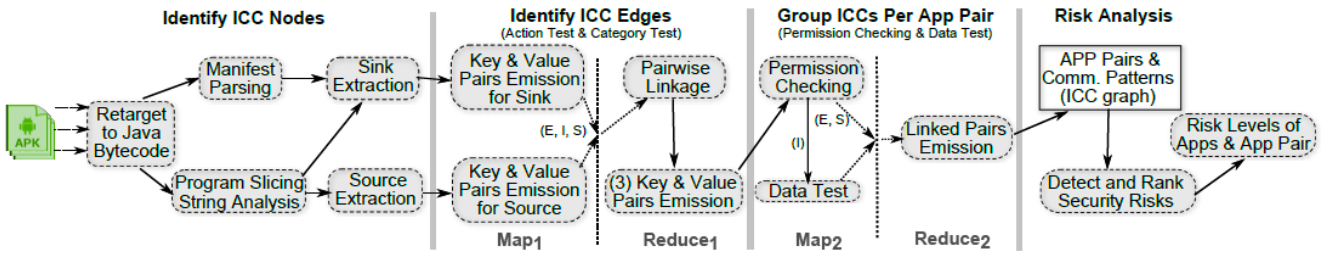


Figure 2: Our Workflow for Analyzing Android App Pairs with MapReduce. The Dashed Vertical Lines Indicate Redistribution

3.3 DIALDroid Workflow

The workflow of our inter-app ICC security analysis involves four key operations: ICC ENTRY/EXIT POINT EXTRACTION, DATAFLOW ANALYSIS, DATA AGGREGATION, and ICC LEAK CALCULATION. They are briefly described below.

- ICC ENTRY / EXIT POINT EXTRACTION: Given an app, we extract the permissions and the attributes of the intent filters from the AndroidManifest.xml file. We perform static analysis to determine the attributes of the intents passing through ICC exit points.
- DATAFLOW ANALYSIS: We use static taint analysis to determine ICC exit leaks and ICC entry leaks in the application. We dynamically adjust the precision configuration of taint analysis to ensure the timely execution of each app.
- DATA AGGREGATION: We aggregate the data extracted in previous two steps to store in a relational MySQL database. DIALDroid database schema is composed of 42 tables and is designed to facilitate efficient storage and fast data retrieval.
- ICC LEAK CALCULATION: We use fine-grained security policies to reduce the search space to look for potential sensitive inter-app ICC channels. Using SQL stored procedures and SQL queries, we compute collusive data leaks and privilege escalations.

DIALDroid executes the first three steps once for each app (complexity O(N), where N is the total number of apps being analyzed). The complexity of ICC leak calculator is O(mN), where m is the number of apps with ICC exit leaks and in the worst case, m = N. However, for real-world apps m would several times smaller than N. In our study, we found m 28 times smaller than N. In the following sections, we provide a detailed description of each of the four subsystems

4 ASSUMPTIONS

In this section, we describe the threat model and the security insights of large-scale inter-app ICC risk analysis. Then we introduce the high-level methodology of our approach.

4.1 MR-Droid Threat Model

Our work focuses on security risks caused by inter-app communications realized through ICCs, covering three most important classes of inter-app ICC security risks.

- **Malware collusion**. Through inter-app ICCs, two or more apps may collude to perform malicious actions [24, 25, 35] that none of the participating apps alone would be able to. The collusion can be realized either by passing Intents to exported components or by using the same *sharedUserId* among the malicious apps. Malware collusion can result in disguised data leak and system abuse.
- **Intent hijacking.** An Intent sent by an app via an implicit ICC may not reach the intended recipient. Instead, it may be intercepted (hijacked) by an unauthorized app. This threat scenario, referred to as Intent hijacking, can be categorized into three subclasses according to the type of the sending component: broadcast theft, activity hijacking, and service hijacking, as introduced in [7]. Intent hijacking may lead to data/permission leakage and phishing attack.
- **Intent spoofing.** By sending Intents to exported components of a vulnerable app, an attacker can spoof the vulnerable (receiving) app to perform malicious actions, including those that are to be triggered only by the Android system. Intent spoofing can be classified into three subclasses by the type of the receiving component: malicious broadcast injection, malicious activity launch, and malicious service launch [7].

4.2 DIALDroid Threat Models

DIALDroid's inter-app security analysis is oriented around identifying pairwise data flows from a sender app A to a receiver app B that result in two types of threats: collusive data leak or privilege escalation. Privilege escalation (aka the confused deputy problem) is a well-defined threat where the receiver app B gains unauthorized permissions or sensitive data as a result of its ICC communications with the sender app A [4, 7]. Although the concept of collusive data leak has been described in the literature [5, 25], it has not been formally defined. In our work, we define collusive data leak as a threat where the receiver app B exfiltrates the sensitive data obtained from its ICC communications with the sender app A to an external destination (e.g., via disk output or network output).

Our analysis aims to detect sensitive data flows (i.e., data flows that carry sensitive information) that result in privilege escalation, collusive data leak, or both. Insensitive data flows and sensitive data flows that do not exhibit collusive data leak and privilege escalation threats are excluded from the analysis. Our threat model excludes intent spoofing, where the sender app forges intents to mislead receiver apps [7]. We consider both explicit and implicit intents.

Next, we first give our definitions for the security terms used in the report, including ICC exit leak, ICC entry leak, and sensitive ICC channel. We then give formal definitions of both privilege escalation and collusive data leak threats. Our experiments further distinguish 6 different subtypes of threats, based on various ICC and security properties.

A sensitive ICC channel refers to an ICC link between two components, from an ICC exit point (i.e., an outgoing ICC such *as startActivity*, *bindService*, and *startActivityForResult*) to an ICC entry point (i.e., an incoming ICC such as *onActivityResult* and *getIntent*) that transfers intents containing sensitive information. Our analysis is focused on sensitive ICC channels and excludes non-sensitive ICC channels.

The property of ICC exit leak is that an app's ICC exit point is data dependent on a sensitive data source, i.e., there exists a dataflow path from the sensitive source to the ICC exit. In the context of inter-app ICCs, we use the ICC exit leak to describe the sender app. Intuitively, the ICC exit leak identifies sender apps that leak sensitive data via inter-app ICCs.

The property of ICC entry leak is that an app's ICC entry point is the source of dataflow paths of sensitive sinks that send the received data externally (e.g., via networks). In the context of inter-app ICCs, we use the ICC entry leak to describe the receiver app. Intuitively, the ICC entry leak identifies receiver apps that leak received data externally. In DIALDroid, our labeling of sensitive source and sink statements follows the SuSi project [46]. Next, we use the terminology introduced above to define privilege escalation and collusion data leak.

Collusive data leak is a threat associated with a sensitive ICC channel between a sender component A in an app and a receiver component B in another app, where A has an ICC exit leak and B leaks the received data from A via an ICC entry leak. Privilege escalation is a threat associated with a sensitive inter-app ICC channel between a sender component A in an app and a receiver component B in another app, where A has an ICC exit leak and B does not have the permission to access the data from A. Collusive data leak may result in privilege escalation. An app may have multiple ICCs. Because of the overlap between the two threats, we further divide them into 6 sub-categories of threats in Table 11. Our empirical analysis follows this fine-grained categorization. The first three types are via explicit intents, while the rest three are implicit intents. Inter-app ICCs that result in neither collusive data leak or privilege escalation are not recorded.

Intentional vs. Unintentional Attacks. One of the difficulties in collusion detection is to confirm the cause of an observed problematic data flow. It is well known that vulnerable sender apps (e.g., with exposed broadcast ICC interfaces) cause privilege escalations [7], i.e., the receiver app can intentionally exploit the vulnerabilities. However, intentional collusion between two apps may also result in privilege escalation. Similarly, collusive data leak may be intentional or inadvertent. Regardless of the causes, these data flows can potentially compromise the device and data security. Our large-scale empirical study helps expose and pinpoint these disguised threats.

Our work is focused on threats associated with inter-app ICCs. We only include intra-app ICC analysis in our benchmark evaluation. All the other results are on inter-app ICCs.

4.3 Security Insights of Large-Scale Inter-app Analysis

In this section, we use an example to illustrate the security differences between single-app analysis and pairwise app analysis. We explain why large-scale pairwise analysis is useful and necessary.

Figure 3 and 4 show parts of two apps, A and B, between which inter-app ICC introduces risks. A has access to location information and sends it out through an implicit Intent. The manifest of B defines an Intent filter that restricts the types of Intents to accept. Once A's Intent passes through the Intent filter, B then sends the received data out through SMS to a phone number (phoneNo).

```
public void onClick(View v) {
 Intent i = new Intent();
 Location loc = locationManager.getLastKnownLocation();
 i.setAction ("android.Intent.action.SENDTO");
 i.setCategory ("android.Intent.category.DEFAULT");
 /*Set the intent data field as location*/
 i.setData(Uri.parse(loc.toString()));
 i.putExtra("PhoneNo.", "*********")
 startService(i);
}
```

Figure 3: Code Snippet of the App A Which Creates and Sends out external ICC Intent Carrying Location Info

```
<service android:name=".SendSMSService" android:enabled="true">
<Intent-filter>
 <action android:name="android.Intent.action.SENDTO"/>
 <category android:name="android.Intent.category.DEFAULT"/>
</Intent-filter>
</service>
public void onStart(Intent Intent, int startId) {
 String pn = Intent.getStringExtra("phoneNo.");
 String msg = "Location: " + Intent.getData();
 /*Send intent data out through text message*/
 smsManager.sendTextMessage(pn, null, msg, null, null);
}
```

Figure 4: The manifest and Code Snippet of the app B Which Processes Received Intents and Sends out Intent Data

With single-app analysis (e.g., ComDroid [7]), B would be identified as always vulnerable to Intent spoofing attacks because its *SendSMSService* component is exported without permission protection (other apps can send SMS through B). Likewise, A would be detected as always vulnerable too, as other apps could hijack the implicit Intent it sends out and get the location information. However, single-app analysis does not consider specific communication contexts. It cannot track the destination of the implicit sensitive Intent **i** in A, and unable to identify which communicating peers may hijack **i** and abuse the sensitive data. The Android design of ICC makes this type of vulnerabilities prevalent. ComDroid [7] reported that over 97% of the studied apps are vulnerable. However, reporting an app as generically vulnerable or malicious is overly conservative, and lead to insufficient precision and excessive alerts.

In practice, the risk becomes real when an app has actual communication contexts with other specific apps. For example, A's security risk increases if malicious apps are found being able to leak A's location information. More malicious hijacking apps trigger higher security risk because users have larger possibilities to install A and the malicious apps at the same time. In addition, A and B also may collude to leak the location information through the implicit Intent, which existing single-app analyses cannot detect both of the apps.

To these ends, we need to evaluate security risks for a given app by not just checking its own ICC properties, but also by carefully examining its external communications with its peers in a neighbor-aware manner. For example, the analysis should be able to track which app may hijack the Intent **i** of A and may collude with this app to abuse the location information; the analysis should give a detailed list of apps (e.g., B) and their corresponding components (e.g., *B.SendSMSService*) involved in the attacks.

The need for the large-scale risk analyses arises from the two main limitations in existing approaches:

- **Large Threats to Validity**. Results are reported only with respect to small sets of sampled apps, and thus suffer potentially large threats to validity (biases) in their findings and/or conclusions.
- **Lesser Security Guarantee.** There is a potentially higher risk of missing true risk warnings because of the limited size of communication context considered (e.g., a highly vulnerable app may be declared safe when only a few external apps are analyzed).

In the rest of this report, we demonstrate how such a solution can be realized by a highly-scalable distributed ICC graph and, a neighbor-aware security analysis. Our solution not only seeks to detect vulnerable apps but also label them with risk levels and rank accordingly. This provides a big picture for the app market to identify the most vulnerable apps to prioritize risk mitigation. Users can be aware of the risk of an app even before installing. The developers can be better guided to fix vulnerabilities when they are aware of the trade-off between functionality and vulnerability.

5 PROCEDURES

5.1 MR-Droid Distributed ICC Mapping

We now present the distributed market-wide ICC mapping algorithms in MR-Droid. The purpose of the algorithms is to build the ICC graph. Our distributed ICC mapping algorithms involve three major operations: Identify ICC Nodes, Identify ICC Edges, and Group ICCs Per App Pair, where the last two operations are performed in MapReduce framework. The entire workflow has many steps and is technically complex. For efficiency, our prototype integrates the tests (on action, category, data, and permission) with other edge-related operations in the last two operations. We describe details of each operation next.

5.1.1 Identify ICC Nodes

The purpose of Identify ICC Nodes is to extract all the sources and sinks from all available apps. We customized the Android ICC analysis tool IC3 [29] for this purpose.

Sink Extraction. We analyze each app's decoded manifest generated with IC3 [13] and retrieve the attributes for sinks listed in Table 1. We also parse the dynamic receivers of each app as exported receiver components.[4] A public component[5] may generate multiple sinks. The number is determined by the number of the component's Intent filters, as each Intent filter is extracted into a sink in ICC graph. A component is extracted into a sink only when it has no Intent filters.

Source Extraction. The attributes of sources are obtained by propagating values in fields of outbound Intents. MR-Droid utilizes IC3 [29] to perform program slicing and string analysis. String analysis may generate multiple possible values for one Intent field, which is due to multiple paths from where the string value is defined. We track all the possible values, and label them with the same identifier. Thus, there may be multiple values associated with an attribute of a source.

[4] The main difference between a dynamic receiver and the components in the Android manifest is that the former can only receive Intents when it is in pending or running status.

[5] A public component is a component that is available for access by other apps via ICCs.

5.1.2 Identify ICC Edges and Tests

Identify ICC Edges and Action/Category Tests[6] operations are performed together in MapReduce. The purpose is to identify all matching source and sink pairs, which are connected with edges in the ICC graph. There are three types of edges: explicit edge, implicit edge, and *sharedUserId* edge, which are shown in Table 2. The explicit edge and implicit edge correspond to explicit and implicit intent. The *sharedUserId* edge corresponds to the private ICC communication using *sharedUserId*.

Table 2: Tests Completed at Different MapReduce Phases and the Edge that the Tests are Applicable to [MR-Droid].

Name	Phase	Apply to
Action Test	$\text{Map}_1, Redcue_1$	Implicit Edges
Category Test	$\text{Map}_1, Redcue_1$	Implicit Edges
Data Test	Map_2	Implicit Edges
Permission Checking	Map_2	All Edges

Our Map_1 algorithm transforms sources or sinks into the <key, value> pairs. Table 3 shows how three types of edges in ICC graph are transformed into <key, value> pairs in Map_1.

During the redistribution phase (after Map_1), the <key, value> pairs that have the same key are sent to the same reducer. Reduce_1 algorithm identifies qualified edges from the redistributed records. Qualification is based on action test and category test for implicit edges and exact string match for explicit and *sharedUserId* edges. These tests are efficiently performed in the redistribution phase with only the edges that pass the action test and category test sent to Reduce_1.

Outputs at the end of Reduce_1 phase are <key, value> pairs, where key consists of package names of a communicating app pair (corresponding to one ICC edge), and value contains properties of the edge.

Table 3: Summary of How Three Types of Edges in ICC Graph are Obtained Based on Source and Sink Information and Transformed into <key, value> Pairs in Map_1 [MR-Droid].

	Key	Value
Explicit Edge	*EFlag, source.type (sink.type), source.destComponent (sink.componentName);*	*hostPackage, source (sink);*
Implicit Edge	*IFlag, source.type (sink.type), source.action (sink.action), source.categoryList (sink.subCategoryList);*	*hostPackage, source (sink);*
SharedUserId Edge	*SFlag. sharedUserId, source.type (sink.type), source.destComponent (sink.componentName);*	*hostPackage, source (sink);*

[6] https://developer.android.com/guide/components/intents-filters.html

5.1.3 Multiple ICCs Per App Pairs

An app pair may have multiple ICC data flows between them. For our subsequent risk analysis, we need to identify and cluster all the inter-app ICCs that belong to an app pair. Therefore, the main purpose of Group ICCs Per App Pair performed in MapReduce is to group together ICCs that belong to the same app pair. In addition, we perform permission checking on all the three types of edges and data test on implicit edges as shown in Table 2. The key is the package names of an app pair. A reducer in the Reduce_2 algorithm records the complete set of inter-app ICCs between two apps that pass the tests.

5.1.4 Workload Balance

The types of actions and categories are unevenly distributed with very different frequencies. This property leads to the unbalanced workload at different nodes in Reduce_1. It highly impacts the performance because most of the nodes are idle while waiting for the nodes with heavy workload. To address this problem, we add a tag before each key emitted by the Map_1. The tag helps to divide the large amount of key-value pairs, which should be sent to one reducer, into m parts feeding m reducers. The tags incur additional communication and disk storage overhead. The optimal parameter m is selected to maximize performance and achieve high scalability. We compare the operation time and computational complexity of our approach with other available inter-app analyses in the evaluation section.

Our distributed ICC mapping conservatively matches the sources with all the potential sinks even if the links are of low likelihood. The full set of links provides security guarantees for our risk analysis. For example, collusion apps may leverage rarely used implicit intent specifications for targeted communication. Ignoring any low-likelihood links would miss detecting such attacks. Our following security analysis incorporates all the possible links and gives a prioritized risk result to minimize security analysts' manual investigation.

5.2 MR-Droid Neighbor-based Risk Analysis

We present our neighbor-aware security risk analysis of inter-app communications, covering the three common types defined in our threat model. Neighbors of each app are the apps connected to it in the ICC graph. The input of our risk analysis is the ICC graph produced by the MapReduce pipeline previously described, and the output is the risk assessment per communicating app pair in the graph for malware collusion threats, or per individual app for (Intent) hijacking and spoofing threats.

For the latter two types of risks for a single app, our neighbor-aware security analysis differs from previous approaches (e.g., ComDroid [7], Epicc [30]), because we examine ICC sinks and sources in the app and all of its inter-app ICC links. Our approach adds more semantic and contextual information to the risk assessment than previous work. Including the neighbor sets as communication contexts reduces false warnings. In addition, our risk assessment identifies the presence of a risk (i.e., detection) and reports how serious that risk may be (i.e., risk ranking). In this section, we explain how we compute the security risks associated with app vulnerabilities and collusions.

5.2.1 Features

To leverage the neighbor sets for a more effective security risk analysis, we extract five key ICC-link features and utilize varying combinations of them for assessing different types of risks, as defined in Table 4. Note that the five features are all expressed as numerical values.

Table 4: Features in Our Security Risk Assessment in MR-Droid.

Feature	Definition	Hijacking	Spoofing	Collusion
Data linkage	Number of outbound/inbound links that carry data	✓		✓
Permission leakage	Number of apps connected to via links that involve permission leaks	✓	✓	✓
Priority distribution	Mean and standard deviation of priority set for outbound/inbound links	✓	✓	
Link ambiguity	Number of outbound explicit links missing package prefix in the target component	✓		
Connectivity	Number of outbound/inbound links and/or number of connected apps	✓	✓	✓

We further explain the definition of each feature as follows: (1) we say that an ICC link carries data if the ICC Intent contains a non-empty data field; (2) a permission leak occurs when an unauthorized app gains access to the resources when it does not have the required permissions for them itself; (3) the priority of a link is a property of the Intent filter for the corresponding ICC Intent; we use median and standard deviation of priority values set for all inbound/outbound links to characterize the distribution; (4) when specifying the target component for an explicit ICC call, the name of the enclosing package of that component may be missing; (5) we consider the number of ICC links and the number of connected apps to quantify link-connectivity and app-level connectivity, respectively.

5.2.2 Hijacking/Spoofing Risk

For a given app, our analysis first computes its risk with respect to individual features, and then aggregates them to obtain an overall risk value. Meanwhile, some features (e.g., priority distribution) are of high importance and thus are used to determine the overall risk level directly without involving other features.

Specifically, for hijacking and spoofing risks, given a feature *f*, the feature-wise risk of *f* for the subject app a is evaluated based on the distribution of all *f* values in the ICC graph, and classified into three risk levels from low to high as illustrated in Figure 5. The rationale is that the risk level is proportional to the feature value and all feature values in the ICC graph are normally distributed. We normalize the categorical value of each feature X by mapping (low, medium, high) to (.1, .5, 1.0) respectively. To incorporate the varying effects of different features on the overall risk, we assign customizable feature weights based on heuristics. The eventual risk value is computed as the weighted sum of feature-wise risk values. We upgrade the risk level if no apps are detected at the higher levels in order to normalize the risk levels. Next, we give details on our risk-evaluation procedures for each risk type.

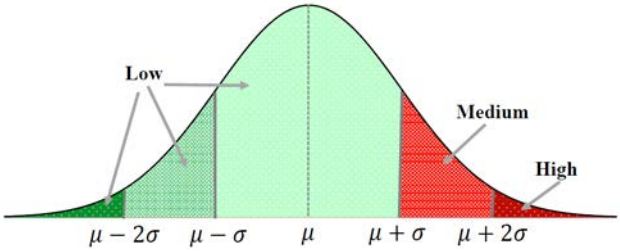


Figure 5: Feature Value Distribution and Classification in MR-Droid.

Hijacking. Since the Intent-sending app is the subject of evaluation (i.e., the app for which risks are assessed) in a hijacking scenario, we consider outbound links only for all relevant features. Also, since an Intent is much less likely to be hijacked if the target is specified explicitly than implicitly, only implicit links are considered. The only exception is for link ambiguity, computing which involves just explicit links. This special feature captures the possibility of hijacking via ambiguous target-component specification---ignoring the package prefix in the target component name. Using these five features, the overall risk value is computed as follows.

- If link ambiguity ≥ 1 or the sum of the two priority distribution parameters is high (mean + standard deviation ≥ 500), report the risk as high and exit.
- Compute the numerical feature-wise risk values for data linkage, permission leakage, app-level connectivity, and link-level connectivity, and sum them up with weights 0.3, 0.4, 0.2, 0.1, respectively. Then, cast the resulting numerical risk value to its categorical risk level, according to where the value falls in the three equal-length subintervals of [0.1, 1.0].

Spoofing. In a spoofing scenario, the Intent-receiving app is the subject of evaluation, thus we consistently consider inbound links only for the three features involved (see Table 4). Also, except for the priority distribution, which counts implicit links only because the priority can only be set for implicit Intents, the other two features consider both implicit and explicit links. Nevertheless, as the spoofing attack is more likely to succeed when using explicit Intents than using implicit ones, we weigh explicit links higher in the composite quantification of the relevant features. Using these three features, the overall risk value is computed as follows.

- If the sum of the two priority distribution parameters is high (mean + standard deviation$\geq$500), report the risk as high and exit.
- Compute the numerical feature-wise risk values for permission leakage, app-level connectivity, and link-level connectivity with respect to explicit links, and sum up them with weights 0.4, 0.2, and 0.2 (total of 0.8), respectively. Then, compute the same three risk numbers with respect to implicit links, and sum up them with weights 0.1, 0.05, and 0.05 (total of 0.2). Finally, add the six risk numbers up and cast the sum back to its categorical risk level, according to where it falls in the three equal-length subintervals of [0.1, 1.0].

5.2.3 Collusion Risk

In collusion attacks, all communicating pairs are analyzed together. The three features involved (see Table 4) count both inbound and outbound links. Also, both implicit and explicit links are used for feature extraction.

MR-Droid looks for the existence of links between the app pair, ignoring the number of links but examining whether the links are explicit or implicit. The analysis uses higher values for features due to explicit links than for those due to implicit ones. The overall risk level is computed as follows.

- Connectivity is quantified as 5 for unidirectional explicit connection (i.e., one app is connected to the other via explicit ICCs yet the other connects back via implicit ones), as 10 for bidirectional explicit connection (i.e., explicit ICCs exist in both directions), and as 3 for bidirectional implicit connection (i.e., no explicit ICC exists between the pair). The data linkage feature is quantified as 3 and 2 for data transfer through explicit and implicit links, respectively. Permission leakage is evaluated as 3 if the leak potentially exists or 0 otherwise.
- Calculate as the aggregate risk value the sum of the three numerical feature values above, and cast the sum into its categorical risk level ranging from low to high, according to where it falls in the three equal-length subintervals of [1,16].
- Identify the apps with low implicit connectivity (empirically, the connectivity is low if the number of inbound or outbound implicit apps is less than 20), and retrieve the corresponding pairs with bidirectional implicit connections. Pairs with both apps having a low implicit connectivity are set as having a high risk of collusion. Pairs with only one app having a low implicit connectivity are set to have a medium risk of collusion.

5.3 DIALDroid Operations

The workflow of DIALDroid involves four key operations: ICC ENTRY / EXIT POINT EXTRACTION, DATAFLOW ANALYSIS, DATA AGGREGATION, and ICC LEAK CALCULATION. They are described as below.

5.3.1 ICC Entry/Exit Point Extractor

This operation identifies all the ICC end points (both entries and exits) from apps, by performing a single pass of analysis on each app. We describe our new tool IC3-DIALDroid for this purpose. We have made this tool open source[7]. Our ICC entry point extractor subsystem extracts the manifest file from the .apk files, parses the permissions requested by the app, and parses the ICC entry points of that app from IntentFilters. We use static analysis to identify intent values, similar to prior studies [29, 30, 38]. Although our implementation uses the libraries provided by IC3, the state-of-the-art ICC extractor [29], our IC3-DIALDroid has several significant engineering enhancements providing much better robustness and much higher intent discovery than IC3. We itemize our main improvements below.

[7] Available at: https://github.com/dialdroid-ndss/ic3-dialdroid.

1. IC3 conservatively adopts call graph generation procedure from FlowDroid skipping incremental callback analysis, which incrementally extends the call graph to include the newly discovered callbacks, and the scan is run again since callback handlers are free to register new callbacks on their own. This process is repeated until the call graph reaches a fixed point [2]. One pass callback analysis improves the runtime performance of IC3. However, it results in missed intents due to imprecise Android lifecycle modeling. IC3-DIALDroid implements incremental callback analysis, which significantly increases the number of discovered intents.
2. IC3-DIALDroid analyzes on Android apks directly. It does not require the Dare tool for reverse engineering [47], and can directly extract the attributes of ICC exit points. In comparison, IC3 is built on top of SOOT [48], a popular Java bytecode analysis framework. Although Android apps are developed in Java, those are compiled into Dalvik byecode (a custom format developed by the Android project) instead of tradetional Java class file. Thus, IC3 requires the Dalvik bytecode to be retargeted using Dare, which not only requires additional preprocessing time, but also may introduce inaccuracies.
3. We identified several defects in IC3, specifically in handling different types of real world apks and the constraint solver’s failure to reach a fixed point even after a long time for some apps. We fixed those defects and implemented code to identify and break race conditions.

We compared the performance of IC3-DIALDroid with IC3 on 29 applications from DroidBench 3.0 and 1000 randomly selected apps with a timeout of 15 minutes for each app. Table 5 shows a comparison between the two tools. On DroidBench, DIALDroid took 13 seconds (8.6%) less than IC3 to compute entry and exit points and identified the same number of intents. On the 1000 randomly selected real-world apps, DIALDroid identified 28% more intents and encountered 33% less failed cases. However, due to more precise lifecycle modeling, IC3-DIALDroid spent 13.3% more time.

Table 5: Comparisons of DIALDroid ICC extractor tool IC3-DIALDroid with the state-of-the-art IC3, in terms of robustness, accuracy, and runtime on benchmark apps and 1,000 real-world apps. Our tool identifies 28% more intents and has 33% fewer failed cases for real-world apps.

	DroidBench 3.0			1,000 Real-World Apps		
	Failed	Intents found	Time	Failed	Intents found	Time
IC3	0	27	151s	**123**	**30,640**	43hrs
Ours	0	27	138s	**83**	**39,080**	48hrs

5.3.2 Dataflow Analyzer

We considered the three state-of-the-art static analysis tools for Android apps, 1) FlowDroid [2], 2) Amandroid [38], and 3) DroidSafe [12], to build a dataflow analyzer. While DroidSafe [12] claims claims to be the most precise static analysis tool, it is 20 to 50 times slower compared to FlowDroid and Amandroid. While both the FlowDroid and the Amandroid offered similar runtime

performances, we found that FlowDroid rarely failed to analyze an app. Therefore, we build our Dataflow analyzer based on the FlowDroid.

We primarily made the following three customizations to improve the performance of the Dataflow analyzer.

1) Number of sources / sinks: Static taint analysis requires a set of sources (e.g., originating methods of sensitive data, such as API calls to retrieve a user's location) and a set of sinks (e.g., methods through which data can exit the application or device). The number of sources/sinks in an app primarily influences the taint analysis time. To manage the number of sources and sinks, the dataflow analyzer analyzes an app in two steps. First, for each of the ICC exit points, we investigate if the intents sent through that point can potentially include any sensitive information (i.e., determine the ICC exit leaks). In this step, the dataflow analyzer uses all the API calls to access sensitive data as discovered by the SuSi project [46] as sources and all methods to initiate ICCs as sinks. Second, for each of the ICC entry points, we investigate if the data extracted from intents can potentially flow out of the application (i.e., determine the ICC entry leaks). In this step, the dataflow analyzer uses the methods to access intent data (e.g., *getIntent*, and *onActivityResult*) as sources and all sinks identified by the SuSi project [46] as sinks.
2) Retry with a less precise configuration: We used two types of configuration for the dataflow analyzer.
    - High-precise configuration: This configuration supports a context-sensitive algorithm with an access path length = 3. In this configuration, the dataflow analyzer builds the complete taint paths. (An access path is of the form a.b.c, where a is a local variable or parameter and b and c are fields. The variable a.b.c has an access path length = 2. An access path length = 0 means a simple local variable or parameter (i.e., in this case a) [2].)
    - Less-precise configuration: This configuration supports a context insensitive algorithm, which does not consider calling context. It is significantly faster, but may have false positives. In this configuration, the access path length is set to 1 and the dataflow analyzer only identifies the sources and sinks, but skips building the complete taint paths.

    By default, the dataflow analyzer runs with the high-precise configuration. However, if a precise-analysis fails to complete within specified time (i.e., 5 minutes), DIALDroid abandons the analysis, and retries with the less-precise configuration.
3) Limit total analysis time: To recover from possible deadlocks, we limit the analysis time for each app to 20 minutes (i.e., if the analysis for an app does not finish within 20 minutes, DIALDroid abandons execution).

5.3.3 Data Module

The data module of DIALDroid aggregates the attributes of an app extracted by the ICC entry/exit point extractor and the dataflow analyzer. The data module stores the aggregated data in a MySQL database. DIALDroid leverages the power of relational databases to encounter scalability issues. Relational databases provide efficient data storage. More importantly, modern relational database management systems facilitate powerful query capabilities to easily transform and retrieve data. DIALDroid uses a highly normalized database schema to efficiently store data and uses indexes on the comparison attributes to support efficient query computation. Our database is composed of 42 tables with a total of 161attributes. A supplementary website[8] provides a detailed diagram of the database schema.

5.3.4 ICC Leak Calculator

We implement the key *calculateSensitiveChannels* procedure inside the database as a SQL stored procedure. This design minimizes potential data transmission delays and leverages the speed, optimization, and efficient queries provided by the database management systems. Because all the inter-app ICC threats in our attack model concern sensitive ICC channels (specifically, requiring ICC exit leaks in sender apps), it is unnecessary to compute ICC links for the intents that cannot possibly contain any sensitive in formation. It drastically reduces the computation complexity.

While matching explicit intents are straightforward, the resolution of an implicit intent involves matching the action, category and data fields with compatible IntentFilter, known in the Android development guide as action test, category test, and data test, respectively. We write SQL queries to compute all the sensitive ICC links originating via implicit intent from a specific app. Due to the complex matching rules, we create two SQL procedures: *categorytest(intent_id,filter_id)* and *datatest(intent_id,filter_id),* which implement the category test and data test, respectively. Queries to compute ICC channels via explicit intents are much simpler.

For computing privilege escalations, we test if the receiver app in a sensitive ICC channel has permissions to access the data transmitted via the carried intent. For computing collusive data leaks, we check if a sensitive ICC channel is joining an ICC exit leak in an app with an ICC entry leak in another app. We show an example query for detecting collusive data leaks in Algorithm 1. In addition, our supplementary website provides scripts to generate all the SQL procedures and the SQL queries[9].

The intent resolution in DIALDroid is based on the libraries provided by IC3. However, in some cases, string analysis in IC3 cannot accurately determine possible values and therefore generates safe over approximated sets (e.g., '.*', a regular expression matching any string constant) [29]. A recent study found that 95% of the ICC links generated by the intents with attributes (i.e., package, component, action, or category) resolved as '.' were infeasible [28]. Therefore, the strict intent matching rules adopted implemented by our ICC leak calculator ignores such over approximated regular expressions. This modification may introduce a few false negatives, but greatly reduces false positives in the subsequent detection.

[8] https://github.com/dialdroid-ndss/dialdroid-db
[9] https://github.com/dialdroid-ndss/dialdroid-db

Algorithm 1. A SQL Query to Detect Inter-App ICC Based Collusive Data Leaks

```
1  SELECT sender.app AS senderapp, idl.method,
       idl.leak_path AS sender_app_path, receiver.app AS
       receiverapp, entryLeaks.leak_path AS
       receiver_app_path, entryLeaks.leak_receiver,
       icc_type FROM SensitiveChannels
2  INNER JOIN Applications sender ON
       SensitiveChannels.fromapp=sender.id
3  INNER JOIN Applications receiver ON
       SensitiveChannels.toapp=receiver.id
4  INNER JOIN EntryPoints ep ON
       ep.class_id=SensitiveChannels.entryclass
5  INNER JOIN ICCEntry_DataLeaks entryLeaks ON
       entryLeaks.entry_point_id=ep.id
6  INNER JOIN ICCExit_DataLeaks idl ON
       idl.exit_point_id=SensitiveChannels.exitpoint
7  LEFT JOIN Intent_Extras ON
       Intent_Extras.intent_id=SensitiveChannels.intent_id
8    -- either no data is passed via putExtra
9    WHERE (Intent_Extras.id IS NULL) or
10   -- OR data is passed via putExtra and receiver path
         also contains the key
11  (Intent_Extras.extra IS NOT NULL AND
       entryLeaks.leak_path LIKE CONCAT
       ('%',Intent_Extras.extra,'%'))
```

6 RESULTS AND DISCUSSION

6.1 Evaluations of MR-Droid

MR-Droid was implemented our system with native Hadoop MapReduce framework. The input is the ICC sources and sinks extracted from individual apps using IC3 [29], the most precise single-app ICC resolution in the literature. We modified IC3 to accommodate the MapReduce paradigm. The Hadoop system is deployed on a 15-node cluster, with one node as master and the rest as slaves. Each node has two quad-core 2.8GHz Xeon processors and 8GB RAM[10].

Datasets. For our evaluation, we apply our system to 11,996 most popular free apps from Google Play. We select the top 500 apps from each of the 24 major app categories (4 apps were unavailable due to bugs in program analysis). We downloaded the apps in December 2014 with an Android 4.2 client. We ran our inter-app ICC analysis and security risk assessment on 12,986,254 app pairs.

In addition to this empirical dataset, we also test our system on DroidBench, the most comprehensive Android app benchmark for evaluating Android taint analysis tools. The latest suite DroidBench 3.0[11] consists of 8 app-pair test cases for evaluating inter-app collusions.

Validation. Because of the lack of ground truth on the empirical data, we devote substantial efforts to manually inspect the apps for validation[12]. In addition, we will evaluate the performance of the distributed ICC analyses by gauging the running time of each phase of the pipeline. Our evaluation seeks to answer the following questions.

[10] The algorithms can also be implemented with Spark [41] with faster in-memory processing. It will require much larger RAMs to hold all the data.
[11] https://github.com/secure-software-engineering/DroidBench/tree/develop
[12] We plan to share these manually labeled datasets as benchmarks to the Android security community.

- Q1: What are the risk levels of app pairs?
- Q2: How accurate is our risk assessment and ranking?
- Q3: What do detected attacks look like?
- Q4: What is MR-Droid's runtime, including per-app ICC resolution and ICC graph analyses?

6.1.1 Results of Risk Assessment

We apply our system to the collected app dataset. The resulting ICC graph contains 38,134,207 source nodes, 26,227,430 sink nodes and 75,123,502 edges. On the per-app level, there are in total 12,986,254 app pairs that have at least one ICC link. Each app averagely connects with 1185 external apps (9.9% of all apps), confirming the overall sparsity of the graph. For non-connected app pairs, we can safely exclude them during the security analysis. Our security analysis focuses on all potential security risks related to Intent hijacking, Intent spoofing and app collusion. We quantify and rank security risks into as categorical risk levels. In total, our system identified 150 high-risk apps, 1,021 medium and 10,825 low risk apps.

Table 6 summarizes the results, highlighting the numbers of apps or app pairs vulnerable to the high and medium level of risks. Prominently, stealthy attacks such as activity hijacking and broadcast theft dominate medium and high risks of any type. These attacks often involve passively steal user data.

Table 6: Numbers of apps found by MR-Droid that are vulnerable to Intent spoofing/hijacking attacks and potentially colluding app pairs reported by our technique, and percentages (in parentheses) manually validated as indeed vulnerable or colluding, per risk category and level. The DroidBench test result is not included in this table.

	Intent Hijacking			Intent Spoofing			
	Activity Hijacking	Service Hijacking	Broadcast Theft	Activity Launch	Service Launch	Broadcast Injection	Collusion Pairs
High	94 (90%)	10 (70%)	15 (90%)	17(100%)	4(100%)	7(100%)	6 (100%)
Medium	790 (80%)	32 (60%)	303 (70%)	9(90%)	8(100%)	0	169(8.3%)
Low	11,112 (25.9%)	11,954 (0%)	11,678 (10%)	11,970 (0.2%)	11,984(0%)	11,989 (0%)	12,986,079(0%)

The more intrusive types of attacks such as service launch and broadcast injection are less prevalent. In addition, considerably collusion-attacks are revealed by our analyses. There are six colluding app pairs are of high risk, and 169 are of medium risk.

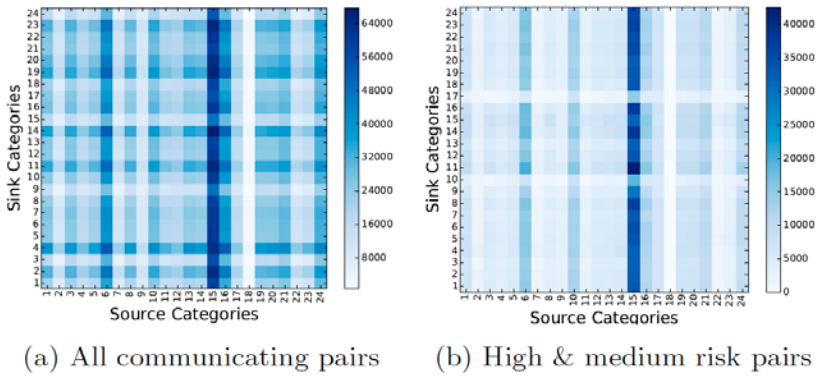
Through risk ranking, we successfully prioritize alerts. Single-app analysis detects the vulnerabilities based on whether the app has exposed its (component) interfaces. In contrast, our approach further examines an app's empirical neighbor set and determines whether it is currently subject to real threats.

App Categories. Our evaluation dataset consists of 11,996 apps across 24 major categories. Each category was selected 500 top free apps (4 apps are excluded due to bugs in program analysis). Detailed statistics and the index of each category is shown in Table 7.

Table 7: App Categories in the Dataset Evaluated by MR-Droid.

Index	Category	# of Apps
1	BOOKS_AND_REFERENCE	500
2	BUSINESS	498
3	COMICS	500
4	COMMUNICATION	500
5	EDUCATION	500
6	ENTERTAINMENT	500
7	FINANCE	500
8	HEALTH_AND_FITNESS	500
9	LIBRARIES_AND_DEMO	500
10	LIFESTYLE	499
11	MEDIA_AND_VIDEO	500
12	MEDICAL	500
13	MUSIC_AND_AUDIO	500
14	NEWS_AND_MAGAZINES	500
15	PERSONALIZATION	500
16	PHOTOGRAPHY	500
17	PRODUCTIVITY	500
18	SHOPPING	500
19	SOCIAL	499
20	SPORTS	500
21	TOOLS	500
22	TRANSPORTATION	500
23	TRAVEL_AND_LOCAL	500
24	WEATHER	500
	Total:	**11996**

Figure 6 depicts the number of app pairs that communicate within or across categories. Figure 4a includes all the 13 million app pairs and Figure 4b shows pairs with at least one app in the high & medium risk lists. We observe that PERSONALIZATION (#15) apps have the most significant contribution to high-risk pairs (Figure 4b). These apps usually help users to download themes or ringtones using vulnerable implicit intents. For example, the app *com.aagroup.topfunny* provides options for user to download apps and ringtones from the web. A malicious app can easily hijack the implicit intent and redirect user to downloading malicious apps or visiting phishing websites. ENTERTAINMENT (#6) apps are also heavily involved high-risk ICCs in similar ways. For example, app *com.rayg.soundfx* also offers downloading ringtones via vulnerable implicit intents. Another high-risk category is LIFESTYLE (#10). These apps often require sensor data (e.g., GPS, audio, camera) for their functionalities. One example vulnerable app is *com.javielinux.andando*. It is a location tracking app but it broadcasts GPS information through an implicit broadcast intent. Other apps can eavesdrop the broadcasted intent and acquire location data even if they don't have location permissions.



(a) All communicating pairs (b) High & medium risk pairs

Figure 6: The Heat Maps of the Number of App Pairs Across App Categories in MR-Droid. The Tick Label Indexes Represent 24 Categories. Detailed Index-Category Mapping in Table 7.

We find that categories such as FINANCE (#7) are less involved in high-risk ICCs. Since these apps often deal with banking and financial payments, it is likely that these apps have put more efforts on security. Even so, we still find high-risk apps under these categories. For example the app *com.ifs.androidmobilebanking.fiid3383* has links in its app to visit its website. However, they trigger implicit intents, which can be easily redirected to phishing websites by malicious apps.

Validation with DroidBench. We have used DroidBench to confirm the effectiveness of our analysis. The latest suite DroidBench 3.0 consists of 8 app-pair test cases for evaluating inter-app collusion. Our approach detected all 8 of them: 6 were labeled as high risk under collusion, and 2 were labeled as high risk under intent hijacking. Our system put the two apps under hijacking category because the way they collude leads to a hijacking vulnerability. They use implicit intents with the default Category and "ACTION_SEND" Action. Our experiment shows that many other apps can receive this type of implicit intents and acquire sensitive information.

6.1.2 Q2: Manual Validation

To further assess the usefulness of MR-Droid, we manually examined over 200 apps to verify the validity of our detection results. More specifically, we randomly select 10 apps from all 7 attack categories at all 3 risk-levels --- for category/level with fewer than 10 apps, we chose all of them. We carefully inspected the selected apps in two ways to check their behavior: (1) static inspection, by which we examine relevant Intent attributes of each app and manually match them against peer apps (in its neighbor set); (2) dynamic verification, in which we run individual apps and app pairs on an Android emulator and observe suspicious behaviors in the activity logs.

The validation result is presented in (the parentheses of) Table 6. For each selected set of apps or app pairs, we report the percentage that was verified to be indeed vulnerable. Overall, this work justified our risk detection and ranking approach. We find apps labeled as high-risk have a much higher rate to be actually vulnerable. We have a 100% true positive rate in detecting collusion, broadcast injection, activity- and service-launch based intent spoofing. We have a 90% true positive rate for activity hijacking and broadcast theft detection. For apps labeled as low risks, the true positive rate is much lower: 5 out of 7 categories have a true positive rate of 0%. These results suggest that the rankings produced by our approach can help users and security analysts prioritize their inspection efforts.

Sources of Errors. Our approach still have a few false alerts (at high-risk level). We find that most of them were caused by unresolved attributes in relevant Intent objects. In those cases, we

conservatively match unresolved source points to all the sinks in order to cover possible ICCs. This matching leads to false positives. One improvement this is to combine our static program analysis with probabilistic models to better resolve ICC attributes [28]. Regarding false negatives, we cannot give a reliable estimation since we did not scan all apps within the market and due to the lack of ground truth.

6.1.3 Q3: Attack Case Studies

Based on our manual analysis, we present a few case studies to discuss empirical insights on ICC-based attacks.

Stealthy Collusion via Implicit Intents. During our analysis, we find some colluding apps also use implicit intents in an effort to avoid detection. Colluding apps usually use explicit intent, since they know “explicitly” which app(s) they are colluding with. However, using explicit intent makes the collusion easier to detect, even by single-app analyses. The new collusion uses highly customized "action" and "category" to mark the implicit intent, hoping no other apps will accidentally interrupt their communication. For example, the app *org.geometerplus.fbreader.plugin.local_opds_scanner* has open interfaces with a customized action name *android.fbreader.action.ADD_OPDS_CATALOG* in its intent filter. Another app *com.vng.android.zingbrowser.labanbookreader* sends implicit intent with the same action and leverages the first app to scan the local WiFi network (the second app itself does not have the permission). This type of collusion cannot be detected if each app was analyzed individually.

Risks of Automatically Generated Apps. We found many of the high-risk apps were automatically generated by app-generating websites (e.g., www.appsgeyser.com). These apps send a large number of implicit Intents, attempting to reuse other apps' functionality as much as possible. For example, *com.conduit.appX39b8211270ff4593ad85daa15cbfb9c6.app* is an automatically generated app and it contains a number of unprotected interfaces including those for viewing social media feeds from Facebook and Twitter. It has 20,805 connections with other apps, five times more than the average.

Hijacking Vulnerabilities in Third-Party Libraries. We observed that a significant amount of vulnerable exit points are from third-party libraries. For example, a flash light app *com.ihandysoft.ledflashlight.mini.apk* bundles multiple third-party libraries for Ads and analytics. One of the libraries *com.inmobi.androidsdk* sends implicit intents to access external websites (e.g., connecting to Facebook). A malicious app can hijack the Intent and redirect the user to a phishing website to steal the user's Facebook password.

Colluding Apps by the Same Developers. We find that colluding apps were usually developed by the same developers. One example is the *org.geometerplus.zlibrary.ui.android* and *org.geometerplus.fbreader.plugin.local_opds_scanner* app pair. The first app is a book reader app with 167,625 reviews and 10 million -- 50 million installs. It leverages the later app (100K -- 500K installs) to scan the user's local network interface. The first app itself does not have the permission to do so. Both of the two apps were developed by “*FBReader.ORG Limited*”.

Another example is the pair of *uda.onsiteplanroom* and *uda.projectlogging*. The first app is for document sharing in collaborative projects, and the second app is for project progress logging. With a click of a button, users will be redirected from the first app to the second app. User's sensitive information in the first app is also sent to the second app without user knowledge. Such information includes user's name, email address, password, etc. These two apps were written by the same developer "*UDA Technologies, Inc.*".

The practical risk of app collisions varies from case to case. The bottom line is, different apps written by the same developer should not open backdoors for each other to exchange user data and access privileges without the user knowledge.

Insecure Interfaces for Same-developer Apps. Usually, apps developed by the same developer have specialized interface to communicate with each other. The secure way to do this is to use *sharedUserID* mechanism to protect the data and interface from exposing to other apps. In our empirical analysis, we find 560 app pairs are developed by the same developer without using *sharedUserID* links. Instead, they use explicit intents to implement the communication interface, which is exposed to all other apps, leaving hijacking vulnerabilities.

6.1.4 Q4: Runtime of MR-Droid

Finally, we analyze the runtime performance of MR-Droid. Figure 7 depicts the time cost of our MapReduce pipeline (y axis) as the number of apps increases (x axis). Overall, the result shows that our approach is readily scalable for large-scale inter-app analysis. The running time of ICC node identification appears to dominate the total analysis cost, yet its growth is linear with the number of apps. In addition, given the sparse nature of the ICC graph (rarely does an app communicate to all apps), we manage to achieve near-linear complexity for edge identification and grouping ICCs. In total, It takes 25 hours to perform the complete analysis on 13 million ICC pairs for 12K apps.

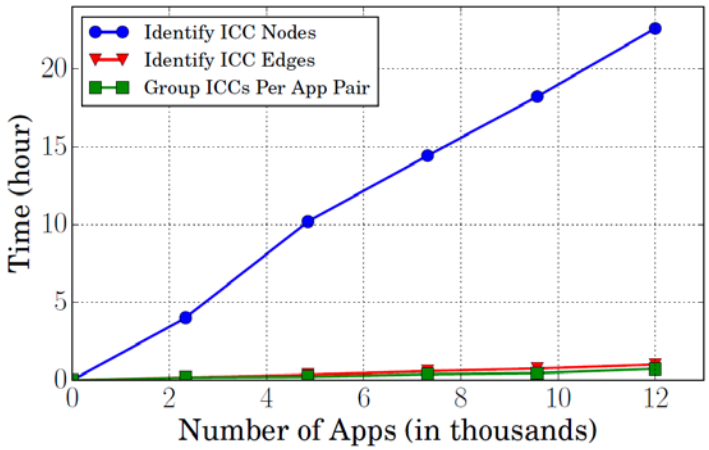


Figure 7: Analysis Time of the Three Phases in MR-Droid.

Noticeably, grouping ICCs for all app pairs only took 44 minutes, rendering 0.0012 seconds per app pair. This speedup benefits from the reduced input size --- a large portion of (unmatched) ICC links have been excluded in the previous phase. Our load balancing design also contributes to speeding up the process. Currently, our cluster has 15 nodes. We anticipate that increasing the cluster size would further speed up the inter-app ICC analysis.

As a baseline comparison, we evaluated the performance of IccTA [21], a non-distributed inter-app ICC analysis system. IccTA needs to first combine two or more apps into one app and then perform ICC analytics. We evaluate IccTA with 57 randomly selected real world apps on a workstation (64GB RAM). It took IccTA over 200 hours to analyze all the apps. We estimate that processing 200 apps with IccTA would take about 18 thousand hours, making it impractical for analyzing market-scale apps.

Complexity Analysis We show operation times and computational complexities of different phases of our approach in Table 8. We denote the average time for identifying ICC nodes for each app

as T, the average time for identifying ICC edges of two apps as t_l, the average time for grouping ICCs per app pair as t_g, the number of available apps as n and the average number of linked apps for each app is m. Note that $t_l + \square_g \ll T$ and $m \ll n$[13]. We also assume that each app has a constant number of entry and exit points on average.

Other inter-app analyses usually run static analysis on two or more apps together (e.g., IccTA coupled with ApkCombiner). Assuming two apps are analyzed each time, the time complexity for parsing n apps is$O(n^2)$. The operation time is $kT\binom{n}{2}$ with 2T being the average analysis time of each two apps.

Table 8: Worst-case Complexity of Different Phases of MR-Droid.

Approaches	Operation Time	Complexity
Identify ICC Nodes	$\mathcal{T}n$	$O(n)$
Identify ICC Edges	$t_l mn$	$O(mn)$
Group ICCs Per App Pair	$t_g mn$	$O(mn)$
Our total (MapReduce)	$\mathcal{T}n + (t_l + t_g)mn$	$O(mn)$

Summary of Findings. Our manual verification confirms the accuracy of our system. For app pairs at high-risk level, we obtain a 100% TP rate for the detection of collusion, broadcast injection, activity- and service-launch based intent spoofing; We have a 90% TP rate for broadcast theft, activity- and service hijacking detection. On the other hand, app pairs at low-risk level indeed have substantially lower TP rate. This result indicates that our risk prioritization is effective. Our empirical analysis reveals new types of app collusion and hijacking risks (e.g., leveraging rarely implicit intents for more stealthy collusion). Our runtime experiments demonstrate that MapReduce pipeline scales well to a large number of apps. Analyzing 11,996 apps and performing ICCs matching took less than 25 hours. More importantly, the runtime cost has a near-linear increase with respect to the number of apps.

6.2 Evaluations of DIALDroid

We aim to answer the following questions through experimental evaluation of DIALDroid.

1. How does DIALDroid compare with other inter-app ICC analysis tools (namely IccTA+ApkCombiner and COVERT), in terms of both detection accuracy and runtime over benchmark apps?
   For conventional intra-app ICC analysis, how does DIALDroid compare with state-of-the-art static taint analysis tools (namely Amandroid, IccTA, DroidSafe, and COVERT), in terms of both accuracy and runtime over benchmark apps?[14]
2. Are there explicit-intent based privilege escalation or collusive data leak pairs? How many cases are via implicit intent based ICCs? Which threat is more common, privilege escalation or collusive data leak?
3. How many problematic app pairs are from the same developer?
4. Are the observed collusive data leaks or privilege escalations are intentional or unintentional?
5. How many apps have ICC exit leaks? How many apps have the ICC entry leaks? What is the distribution of sensitive ICC channels across app categories?

[13] The ICC graph is sparsely connected.
[14] We choose not to include DidFail [18], because a prior study reported poor performance of DidFail [21].

6. What are the top 10 leaked permissions in privilege escalation cases? What categories of Google Play apps cause the most collusive data leaks?
7. How long does DIALDroid take to analyze hundreds of thousands of real-world Android apps?

In addition to answering the above questions, we also released a benchmark consisting of real world apps for comparing the detection capabilities for collusive data leaks. Unless specified, experiments were conducted on a Dell Tower Precision 7810 workstation running Ubuntu 14.04LTS 64bit with 16 core Intel Xeon 2.4GHz CPU, 64GB RAM, and an SSD drive. We enlisted four virtual machines for the large-scale experiment in 4.8.

We evaluate both real-world apps and benchmark suites. Our three datasets are described below.

- Dataset I (Benchmarks). We evaluate the following benchmarks.
  DroidBench 3.0: DroidBench is the most comprehensive benchmark suite to evaluate the effectiveness of Android taint analysis tools. Among the 174 test cases provide by the DroidBench 3.0[15], 10 test-cases aim to evaluate intra-app leaks and 11 test-cases aims to evaluate inter-app collusions.
  DroidBench (IccTA): IccTA introduced 23 test cases for intra-app leaks and 6 test-cases for inter-app leaks and are available in the IccTA branch[16] of the DroidBench.
  ICC-Bench: ICC-Bench[17], introduced by Amandroid [38], provides 11 test cases for Intra-app leaks. While ICC-Bench did not mention about inter-app leaks, we found and verified 9 inter-app leaks in ICC-Bench.
  For inter-app ICC analysis, our comparison is on 20 inter-app ICC test cases from these benchmark suites. We also evaluate 44 intra-app ICC test cases for completeness. A test case may contain multiple ICC leaks.
- Dataset II (Google Play apps). Dataset II consists of 100,206 most downloaded Android apps (as of June, 2015) belonging to 16 popular categories from Google Play. Table 9 shows the distribution of the apps across the categories.
- Dataset III (All real-world apps). Dataset III consists of all apps from Dataset II as well as 9,944 malware apps fromVirus Share database (https://virusshare.com/). Dataset III has a total of 110,150 apps.

[15] https://github.com/secure-software-engineering/DroidBench/tree/develop
[16] https://github.com/secure-software-engineering/DroidBench/tree/iccta
[17] https://github.com/fgwei/ICC-Bench

Table 9: Statistics of our program analysis during the scalability evaluation of DIALDroid with 110,150 real-world apps. High/low precise config. refers to the high or low precision configuration of FlowDroid used for the static taint analysis of an app, respectively. Timeout refers to the percentage of apps that FlowDroid cannot finish after 20 minutes under the low precision configuration.

Category	# Analyzed	High-precise config.	Less-precise config.	Time-out
Books & Reference	8,146	83.7%	13.8%	2.5%
Business	5,949	72.7%	18.0%	9.3%
Comics	2,057	80.5%	16.7%	2.8%
Communication	5,632	77.5%	13.1%	9.4%
Entertainment	8,189	77.4%	16.7%	5.9%
Lifestyle	7,368	76.6%	17.5%	5.9%
Medical	1,801	81.5%	13.6%	4.9%
Personalization	7,435	84.7%	14.1%	1.2%
Photography	8,041	79.5%	16.6%	3.9%
Productivity	7,582	82.6%	12.8%	4.6%
Shopping	6,336	77.1%	15.3%	7.6%
Social	6,870	71.4%	20.8%	7.8%
Sports	7,047	78.1%	16.2%	5.7%
Tools	8,105	85.0%	12.1%	2.9%
Transportation	6,323	81.9%	12.7%	5.4%
Travel & Local	3,344	73.8%	18.5%	7.5%
Virus Share	9,944	63.7%	26.5%	9.8%
Total*:	110,150	83.6%	10.7%	5.7%

**A few apps belong to multiple categories.*

Our implementation uses FlowDroid for static data-flow analysis, a customized ICC resolution tool for string analysis of intents, and a MySQL database for ICC linking and detection. Our sensitive sources and sinks definitions follow SuSi [46]. In Section 5.4, we discuss how sensitive source and sink definitions may impact the detection.

6.2.1 Inter-app ICC BenchMark

Table 10 shows the benchmark comparison results of our inter-app ICC analysis. DIALDroid has the highest precision[18] (91.7%), the highest recall[19] (95.6%), and the highest F-measure (0.93) among the three tools.

IccTA performed poorly (8.7% recall), mainly because ApkCombiner was unable to combine the majority of the app pairs (62%). For the successfully combined apks, IccTA can only detect the inter-app leaks that are in DroidBench-IccTA (i.e., the benchmark that was developed by the same authors). Due to inaccurate intent resolution, COVERT reported a high number of false positives (323). COVERT failed to detect all inter-app leaks from the DroidBench 3.0.

We performed manual inspection on our failed cases. Among the 21 inter-app pairs, nine lead to privilege escalation. DIALDroid was able to detect six of those with a 100% precision and 66.7% recall. DIALDroid failed to report transitive (indirect) privilege escalations (i.e., data leaked via an intermediate component with the same level of permissions as the source component). In contrast, COVERT failed to report any of those nine privilege escalations.

[18] Precision is the percentage of identified cases that are true leaks.

[19] Recall is the percentage of present leaks that are detected.

We compare the inter-app analysis runtime of COVERT, IccTA+ApkCombiner, and DIALDroid, with 57 randomly selected apps from Google Play Market. Out of the 1,596 pairs, ApkCominer was able to combine only 501 pairs (31%) and IccTA took 203 hours to complete on the combined apps. COVERT ran for 26 hours and then crashed during the formal model generation step [3]. In comparison, DIALDroid took 6.1 hours to complete. It only abandoned two apps, as DIALDroid was unable to finish within 20 minutes in each case.

6.2.2 Threat Breakdown for Dataset II

We break down the threats into six disjoint categories, which are listed as threat types I to VI in Table 11. The categories are disjoint in that an inter-app ICC belongs to one and only one category. Some sender apps may appear in ICCs of multiple categories.

We then run DIALDroid on Dataset II (Google Play apps). For each threat type, we summarize our findings in Table 11. Because Google Play market is known to deploy app vetting mechanisms (e.g., Google Bouncer), it is reasonable to assume the apps in Dataset II have passed some single-app screenings.

We found no collusive data leaks or privilege escalations based on explicit intents, i.e., no inter-app ICCs of Threat Types I to III. This result suggests that explicit intent based collusion is very rare. (They might exist, but are out of the scope of our dataset.) Therefore, collusion analysis needs to be focused on implicit intents based ICCs, as opposed to explicit intents.

For inter-app ICCs via implicit intents, we distinguish three cases: both collusive data leak and privilege escalation in Threat IV, privilege escalation without collusive data leak in Threat V, and collusive data leak without privilege escalation in Threat VI. We highlight some key results next. The most severe threat type is Threat IV, where collusive data leak and privilege escalation occur simultaneously. We found 33,756 app pairs originating from 51 sender apps that exhibit both collusive data leak and privilege escalation behaviors via implicit intents. Because of the sensitive data from the sender app is leaked externally by the receiver app and the receiver app is under the disguise of having fewer permissions, apps in Threat IV is the most serious.

It is not surprising that we observe a huge number (5,638,772) of inter-app ICC channels with the privilege escalation threat in Threat V. Some app pairs may have multiple ICC channels between them. Interestingly, these 5,638,772 ICCs with privilege escalation threat originate from only 134 problematic sender apps.

For Threat Type VI (collusive data leak without privilege escalation), we found 14,433 such app pairs originating from 38 sender apps. That is, these app pairs exhibit collusive data leak behaviors; however, the receiver apps do not gain new permission privileges, i.e., the receiver apps have the authorization to access the received data. In addition, we found that a large number of sender apps in Threat Type VI are also sender apps in Threat Type IV. We performed case studies for each of the Threat Types IV, V, and VI. Some cases in Threat Type VI suggest that the collusive data leaks are unintentional.

Table 10: Comparisons on DIALDroid inter-app ICC analysis with DroidBench 3.0, DroidBench (IccTA branch), and ICC-Bench. Multiple circles in one row means multiple inter-app collusions expected. An all-empty row: no inter-app collusions expected and none reported. † indicates that the tool crashed on that test case.

✓= a correct warning, *= a false warning, ◯= a missed leak, Ⓟ= a privilege escalation reported, ‡= did not test or N/A.

Source App	Destination App	# ICC Exit Leaks (Dest.)	# ICC Entry Leaks (Sink)	Privilege Escalation	COVERT	IccTA + ApkCombiner	DIALDroid (Ours)
DroidBench 3.0							
SendSMS	Echoer	1	3	✓	◯◯	◯◯	✓✓Ⓟ
StartActivityForResult1	Echoer	2	3	✓	◯◯	◯◯	✓✓Ⓟ
DeviceId_Broadcast1	Collector	2	1	✓	◯	◯†	✓*Ⓟ
DeviceId_ContentProvider1	Collector	2	1	✓	◯	◯†	✓
DeviceId_OrderedIntent1	Collector	3	1	✓	◯	◯†	✓
DeviceId_Service1	Collector	1	1	✓	◯	◯†	✓Ⓟ
Location1	Collector	2	1	✓	◯	◯†	✓Ⓟ
Location_Broadcast1	Collector	3	1	✓	◯	◯†	✓*Ⓟ
Location_Service1	Collector	2	1	✓	◯	◯†	◯
Incorrect app pairings					(172 *)	‡	
DroidBench (IccTA branch)							
startActivity1_source	startActivity1_sink	1	2		✓	✓	✓
startSevice1_source	startService1_sink	1	2		✓	✓	✓
sendbroadcast1_source	sendbroadcast1_sink	1	2		✓	✓	✓
Incorrect app pairings					(104 *)	‡	
ICC-Bench							
implicit1	implicit_src_sink	1	1		✓	◯†	✓
implicit1	implict_nosrc_sink	1	1		✓	◯	✓
implicit5	implicit6	1	1		◯	◯†	✓
implicit6	implicit5	1	2		✓	◯†	✓
implicit_src_nosink	implicit_src_sink	1	1		✓	◯	✓
implicit_src_nosink	impliict_nosrc_sink	1	1		✓	◯	✓
implicit_src_nosink	implicit1	1	1		✓	◯	✓
impilicit_src_sink	implicit1	1	1		✓	◯†	✓
impilicit_src_sink	implicit_nosrc_sink	1	1		✓	◯	✓
Incorrect app pairings					(47 *)	‡	
Sum, Precision, Recall, and F measure							
True positive (✓), higher is better					11	3	22
False positive (*), lower is better					323	0‡	2
False negative (◯), lower is better					12	20	1
Precision, p = ✓/(✓+*)					**3.3%**	**100%‡☛**	**91.7%**
Recall, r = ✓/(✓+◯)					**47.8%**	**8.7%**	**95.6%**
F-measure = $2pr/(p+r)$					**0.06**	**0.16**	**0.93**

☛ Since we were unable to execute IccTA+ApkCombiner on most of the pairs, it's precision value is misleading and does not reflect it's actual performance.

Table 11: Summary of problematic inter-app ICC channels in each threat category. Sender apps and receiver apps are from Google Play Market. All the ICC channels shown are sensitive with ICC exit leaks in the sender app.

Categorization				Results			
Threat Type	Collusive Data Leak	Privilege Escalation	Intent Type	# of Distinct Source App	# of Distinct Receiver App	Total ICC Channels	Total App Pairs
I	Yes	Yes	Explicit	0	0	0	0
II	No	Yes	Explicit	0	0	0	0
III	Yes	No	Explicit	0	0	0	0
IV	Yes	Yes	Implicit	51	1,795	144,995	33,756
V	No	Yes	Implicit	134	81,234	5,638,772	3,181,576
VI	Yes	No	Implicit	38	1,044	68,513	14,433

6.2.3 Case Studies

Threat TYPE IV [collusive data leak & escalation]: App *com.koranto.mkmn*. This sender app is to provide prayer times for Muslims around the world. The MainActivity of *com.koranto.mkmn.activities* retrieves the user's location (i.e., *getLastKnownLocation*, permission=*android.permission.ACCESS_FINE_LOCATION*) and sends it via an implicit intent (action=*android.intent.action.SEND*, MimeType=*text/plain*).

This app communicates with 35 receiver apps and the resulting 35 inter-app ICCs exhibit both collusive data leak and privilege escalation behaviors. For example, *br.com.coderev.acumapa*, which provides an acupuncture map overlaid on the image captured by the device camera, can receive this intent and write the retrieved location information to a file.

Threat TYPE V [escalation w/o collusive data leak]: App pair *riverside.gov.NatureSpotter* → *com.dnsdojo.mokkouyou.android.image*. The sender app is to help nature lovers record their observations. *riverside.gov.NatureSpotter* retrieves the user's latitude and longitude (i.e., *getLatitude*/*getLongitude*, permission=*android.permission.ACCESS_FINE_LOCATION*) and sends the location and an image via an implicit intent (action=*android.intent.action.SEND*, MimeType=*image/jpg*). The receiver app, which is an image resizer, defines an intentFilter to accept the above intent. However, *com.dnsdojo.mokkouyou.android.image* does not have the permission to access user's location, this ICC communication leads to escalated privileges.

Threat TYPE VI [collusive data leak w/o escalation]: *com.glympse.android.glympseexpress*. The sender app enables a user to share a temporary link, so that others can see the user's locations on a live map. *com.glympse.android.glympseexpress.Util* retrieves the user's country, SIM operator, and network operator and sends the information via an implicit intent ((i.e., method=*getSimSerialNumber/getNetworkOperator*, permission=*android.permission.ACCESS_PHONE_STATE*, action=*android.intent.action.SEND*, MimeType=*text/plain*).

We found that 809 receiver apps that have permissions to access phone state, are able to accept that intent, and extract data sent via *android.intent.extra.TEXT* field. For example, *com.twr.twr_android.app*, which is a hyper-local chat and proximal discovery app, can receive this intent and leak the retrieved information to a log.

It is impossible to learn the true intentions behind these implicit intent based collusive data leak or privilege escalation behaviors. Is the developer's intention malicious (e.g., for deliberately evading detection or stealing sensitive data) or benign (e.g., due to poor programming practices)? We further discuss the security implications in Section 5.4.

6.2.4 ICC Exit and Entry Leaks

For Dataset III, the number of sender apps with ICC exit leaks is an order of magnitude fewer than the number of receiver apps with ICC entry leaks. Specifically, DIALDroid identified a total of 34,991 ICC exit leaks that are caused by 4,035 sender apps (3.66% of the total apps). DIALDroid identified a total of 249,263 ICC entry leaks that are caused by 32,855 receiver apps (29.82% of the total apps).

Out of the 4,035 sender apps with ICC exit leaks, 2,160 of them (≈ 1.96% of total apps) initiate sensitive ICC channels. Although it does not necessarily mean that the remaining apps are threat-free, as they may communicate with apps outside of our dataset, the number of problematic sender apps is

somewhat surprisingly small. However, because of the use of implicit intents in the inter-app ICCs, these 2,160 sender apps generate millions of ICC links.

Figure 8 shows the percentages of leaking apps out of each app category. For Google Play apps, Personalization has the highest percentage of apps with ICC exit leaks (in sender apps), which is only slightly lower than the Virus Share category. For ICC entry leaks in receiver apps, the percentages are rather high across all the Google Play app categories, with Photography and Business being the highest.

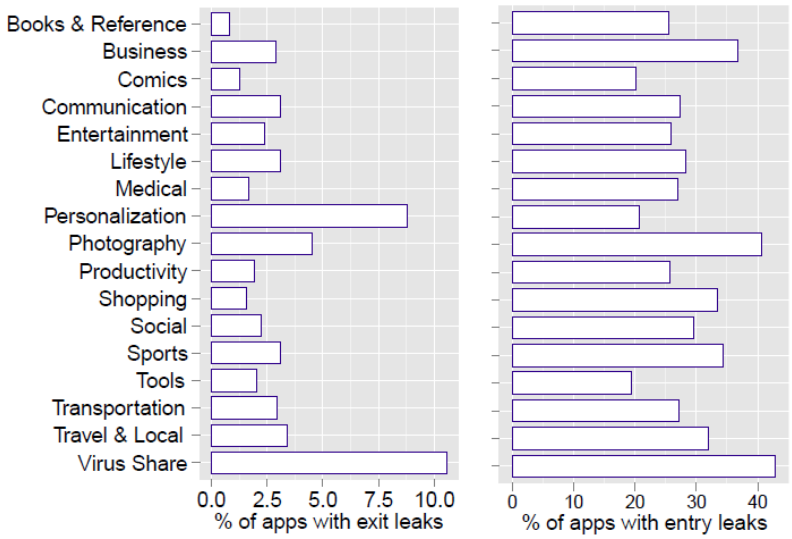


Figure 8: Percentages of Apps out of Each App Category Have ICC Exit Leaks (left) or ICC Entry Leaks (right) in Dataset III by DIALDroid.

6.2.5 Sensitive ICCs

For Dataset III, DIALDroid found 9,579,544 (≈ 9.5 million) potentially sensitive ICC channels. Most of the (≈ 99.7%) sensitive ICC channels are inter-app, and the rest are intra-app. These sensitive ICC channels originate from only 2,160 apps.

Table 12 shows how the sender apps involved in sensitive ICC channels or collusive data leaks are distributed across different app categories for Dataset III. Intuitively, this table summarizes the problematic sender apps and their categories. We highlight the categories with at least one percentage over 9%.

Table 12: The Distribution of Sensitive ICC Channels and Collusive Data Leaks Among App Categories for Dataset III by DIALDroid. An App May Have Multiple Sensitive ICC Channels.

Category	% of total Apps	% sensitive ICC channels (origin)	% collusive data leaks (origin)
Books & Reference	7.40%	6.29%	1.26%
Business	5.40%	3.81%	2.52%
Comics	1.87%	1.27%	1.26%
Communication	5.11%	1.53%	0.03%
Entertainment	7.43%	7.67%	**10.60** %
Lifestyle	6.69%	0.03%	0.12%
Medical	1.64%	1.27%	1.26%
Personalization	6.75%	**10.02**%	**10.21** %
Photography	7.30%	6.33%	6.07%
Productivity	6.88%	0.01%	0.46%
Shopping	5.75%	1.26%	1.20%
Social	6.24%	3.15%	4.85%
Sports	6.40%	2.52%	4.64%
Tools	7.36%	4.12%	**10.27** %
Transportation	5.74%	**9.07** %	3.78%
Travel & Local	3.04%	**12.76** %	1.27%
Virus Share	9.03%	**28.90** %	**42.69** %

For Google Play apps, Travel & Local apps initiate the most sensitive ICC channels, 12.76%, which is most likely due to passing the user's location information to another app. In contrast, this category has a relatively low percentage of collusive data leak cases, which indicates the location or other sensitive information being passed is likely consumed by the receiver app, as opposed to being leaked via disk output or network output. Transportation apps follow a similar pattern. Personalization and entertainment categories have high percentages of problematic sender apps for both types of inter-app ICC threats.

In comparison, sender apps from Virus Share are involved in a substantially higher number of detected sensitive ICC channels and collusive data leaks, which is expected. Although they account for 9.03% of the apps in Dataset III, 28.9% of the sensitive ICC channels and 42.69% of the collusive data leaks are originated from apps in Virus Share. The high percentage (42.69%) of collusive data leaks originating from malware apps indicates that malware apps actively seek and transfer sensitive information.

Intra-app ICC leaks. Among the $\approx$ 9.5 million problematic ICC links, only 26,047 ($\approx$0.27%) were intra-app ICC links. DIALDroid identified only 150 of those intra-app links belonging to 20 different apps can potentially cause ICC leaks. Prior research has focused mostly one identifying intra-app ICC leaks [2, 12, 24, 21]. However, our results suggest that the number of intra-app ICC leaks are negligible compared to inter-app ICC problems.

6.2.6 Permission and Method Distributions

Table 13 shows the number of different permissions leaked via all privilege escalation scenarios for Dataset III. Recall that Dataset III includes Google Play apps and apps from Virus Share. The results suggest that user's location, device information, and current cellular network information are overwhelmingly more likely to be transferred to apps that do not have corresponding access

permissions. The permission ACCESS_NETWORK_STATE gives the app authorization to access NetworkManager to monitor network connections, which is useful for device fingerprinting. Similarly, the permission ACCESS_WIFI_STATE provides the access to WifiManager and can be used for fingerprinting.

Table 13:Top 10 Permissions Leaked via Privilege Escalation in Dataset III by DIALDroid.

Permission	# Cases
android.permission.ACCESS_FINE_LOCATION	3,651,927
android.permission.ACCESS_COARSE_LOCATION	3,651,927
android.permission.READ_PHONE_STATE	1,578,455
android.permission.ACCESS_WIFI_STATE	509,380
android.permission.PACKAGE_USAGE_STATS	425
android.permission.ACCESS_NETWORK_STATE	239
android.permission.BLUETOOTH	143
android.permission.MANAGE_ACCOUNTS	10
android.permission.BLUETOOTH_ADMIN	1
android.permission.READ_CALENDAR	1
Total:	9,373,348

Table 14 (first two columns) shows the most common sensitive source methods in collusive data leak cases in Dataset III. Methods to retrieve users' location (i.e., *getLastKnownLocation*, *getLatitude*, and *getLongtitude*) are the most common sources of ICC leaks. Other common sources include methods to retrieve information that can uniquely identify a user (i.e., *getDeviceId, getSubscriberId, getSimSerialNumber* and *getLineNumber*). Similarly, Table 14 (last two columns) shows the most common sensitive sink methods. SharedPreferences and Log are the mostly used for collusive data leaks. Other APIs are related to file, network, and SMS. In Section 5.4, we discuss how relaxing sensitive sourcesink definitions impacts the results.

Table 14: Sensitive sources/sinks involved in collusive data leaks in Dataset III by DIALDroid.

Sensitive Source	%	Sensitive Sink	%
getLastKnownLocation	40.5%	android.content.SharedPrefs	49.0%
getDeviceId	15.9%	android.util.Log	48.3%
getConnectionInfo	8.5%	java.io.OutputStream	1.1%
getSubscriberId	8.3%	java.net.URL	0.9%
getCountry	6.7%	java.io.FileOutputStream	0.7%
isProviderEnabled	4.3%	org.apache.http.HttpResponse	0.1%
getLatitude	3.8%	android.telephony.SmsManager	0.03%
getLongitude	3.8%		
getSimSerialNumber	1.8%		
getSimOperatorName	1.7%		
getLine1Number	1.6%		
getNetworkOperator	1.6%		
others	1.5%		

6.2.7 Runtime on 110K Apps

For scalability evaluation, we measure how long DIALDroid takes to analyze our largest dataset, Dataset III with 110,150 apps. We used four virtual machines, each with 4 processor-cores, 64GB RAM, and 1 TB hard drive to analyze the apps. We stored the results to a MySQL database hosted on a server with an eight-core processor and 80GB RAM. The ICC Leak Calculator module of DIALDroid computed all the sensitive ICC channels among the 110,150 apps in 82 minutes. This

computation is fast, because although the total number of ICC links is huge, the percentage of sensitive ones is extremely low (about 0.57% as estimated by our experiment[20]. Non-sensitive entries are not touched in the computation. Our relational database schema is efficient and consumes only 6.3 GB space for storing the information for 110,150 apps.

DIALDroid was able to analyze more the 80% of the apps within five minutes. The average analysis time per app was 3.45 minutes. Figure 9 shows the distribution of analysis time for the applications. Adding the individual analysis time for each app (i.e., as if all the apps were analyzed on a single machine), DIALDroid took a total of 6339.6 hours to analyze the 110,150 apps. DIALDroid was able to complete 83.6% of the apps with a high-precise configuration within five minutes. For 10.7% of the apps, DIALDroid timed out in high precise configuration but was able to analyze successfully within five minutes when retried with a less precise configuration. For the remaining 5.7% of the apps, DIALDroid failed to complete the analysis within the specified execution limit (20 minutes). Table 12 shows statistics of the apps and our program analysis.

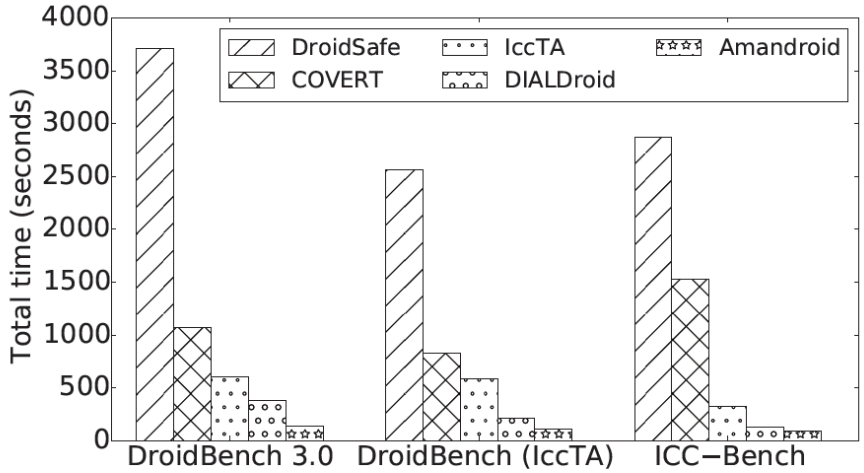


Figure 9: Comparisons of the averaged intra-app execution time of DIALDroid on single-app benchmarks with four other state-of-the-art solutions.

[20] We first computed all the possible ICC links originating from 1,000 randomly selected applications and obtained ≈ 21.8 million ICC links originating from those 1000 apps. Among those ≈ 21.8 million ICC links, only 124K (≈ 0.57%) ICC links were sensitive. The rest of them do not carry sensitive data.

6.2.8 New Benchmark Released

In order to validate the detected collusion pairs and privilege escalations, we inspected the taint paths reported by DIALDroid. We further validated the leaks through manual inspections on the code. We converted the .apk files to .jar files using the dex2jar[21] tool. We decompiled the .jar files to Java source code using a Java decompiler[22]. We manually inspected the source codes to verify leaks. Based on our manual verification, we have compiled a benchmark suite, DIALDroid-Bench[23], to test inter-app collusion. Currently, the suite contains 30 real-world apps from the Google play. To our knowledge, this is the first such benchmark using real-world apps, as opposed to proof-of-concept apps.

6.3 Security Recommendations

Based on our empirical study and manual verification, we make the following recommendations for app developers to reduce potential security risks.

- When carrying out sensitive operations, developers are encouraged to use explicit Intents instead of implicit ones. For example, to access Facebook account, communicating to the Facebook app via explicit Intents is preferred over a https URL as an implicit Intent to the Facebook webpage.
- When possible, it is recommended to integrate all necessary functionalities into a single app. If the developer has to develop multiple apps that communicate with each other, it is encouraged to use the safe communication via *sharedUserID* to restrict other apps accessing the interfaces.
- Developers are encouraged to use customized actions and to enforce data and permission restrictions. These customized configurations (compared to using the default) will greatly reduce the chance to accidentally communicate with other apps, hence reducing risks.
- It is recommended to avoid automatically generating apps using tools or services. Developers should be aware that third-party libraries could be vulnerable or leak user's sensitive information.

6.4 Discussion

Unintentional leaks and escalations. Although the reported collusive data leak and permission escalation cases may be unintentional (e.g., due to insecure design or poor development practices), these apps still pose threats to user’s sensitive data and device. Several lessons can be learned by developers in order to prevent or reduce such threats. ICC sender apps should avoid transferring sensitive data through Activity or Service based implicit intents. Permission checking is needed for Broadcast intent with sensitive information. Whenever possible, explicit intents are preferred for communicating sensitive

[21] https://github.com/pxb1988/dex2jar
[22] http://jd.benow.ca/
[23] https://github.com/dialdroid-android/dialdroid-bench/

data between apps. For receiver apps, enforcing strict restrictions for each entry point (e.g., add pathPattern in intentFilter) reduces unintended and unexpected ICCs.

Sensitive source and sink definitions. The choice of sensitive source and sink impacts the number of reported ICC anomalies. A smaller set of sensitive sources and sinks generates a smaller number of alerts. For example, as shown in Table 14, *android.util.Log* accounts for 48.3% of the sensitive sinks (in receiver apps) in the detected ICC leaks. When excluding both Log and *SharedPreferences* from the sensitive sink list, our query returns a much reduced number (15,109) of collusive ICC links.

Our sensitive sources and sinks definitions follow SuSi [46], which includes Android logging and *SharedPreferences*. In the exploits (e.g., LogCat and CatLog) are still possible. Thus, our evaluation includes logging as a sensitive sink in our evaluation. *SharedPreferences* are key-value pairs maintained by the Android system. An app can read and write the value associated with the key. There are three modes for *SharedPreferences:* private (i.e., accessible only by the owner app), world-read (i.e., others can read), and world-write (i.e., others can read and write). In virtually all taint analyses, *SharedPreference* is labeled sensitive. Even if it is configured as private, other components of the same app can access it resulting in sensitive data flows.

The advantage of DIALDroid is that its database backend allows security analysts to easily adjust and customize sensitivity definitions to refine query results.

App chain length. Three or more apps can possibly create a chain of ICC links to leak data. For example, three apps, A, B, and C, create an ICC chain, where app A transfers sensitive information to app B via an ICC exit leak; app B then leaks that information to app C. A chain of three or more apps is a special case of two app-based ICC collusion, where the receiver app leaks data extracted from an intent by initiating another ICC (i.e., ICC entry leak with ICC initiation methods as sink). Therefore, DIALDroid and MR-Droid report A→B link described above as an inter-app collusion.

Among the three benchmarks evaluated, DIALDroid identified following two scenarios in the ICC-Bench, where three components work together to leak sensitive information.

1. implicit5.M ainActivity → implicit5.F ooActivity → implicit5.H ookActivity
2. implicit6.M ainActivity → implicit5.F ooActivity → implicit5.H ookActivity

Although we did not find any chain of more than two components among the 110K real-world apps, DIALDroid is capable of identifying such chains.

User applications. Although DIALDroid is for marketplace owners, Android users can also benefit from this tool. For example, enterprise users can check possible inter-app collusions using DIALDroid before allowing certain apps on the devices of their employees. Moreover, a large-scale public database similar to ours, when regularly updated, can be queried by users to find out possible inter-app communications to or from a particular app.

Limitations. Similar to most other approaches based on static analysis, our approach shares some inherent limitations. For example, DIALDroid can resolve reflective calls only if their arguments are string constants. Since our strict intent matching rules ignore over approximated regular expressions, DIALDroid may fail to compute some ICC links.

DIALDroid loses field identification. DIALDroid uses a regular expression string search within the ICC entry leak path for the source data keys. As we encountered in *startActivity6* test case, this search may return false positives if the path contains any string that contains the key as a substring. We manually inspected 30 taint paths from real-world collusion pairs identified in our study and did not observe any such occurrences.

To enable large-scale analysis, we limited our analysis time per app. Although DIALDroid failed to analyze only 5.7% of the applications within allocated time (i.e., 20 minutes), there is a possibility that some of those applications could cause collusions.

*Alternative Implementations of MR-Droid***.** Our ICC mapping algorithm is designed under the universal MapReduce programming model. Any platforms that are built on the MapReduce programming model can implement our algorithm. For example, our system can easily fit in Spark with "**flatMap**" and "**groupByKey**" functions for faster in-memory processing. However, it will require the cluster to have very large RAMs to batch process all the apps in memory.

One way to further speed up our system is to use GPU (CUDA). In practice, GPU (CUDA) is for computation only and orthogonal to MapReduce. GPU alone is not enough for large-scale data processing, but could be integrated with MapReduce to achieve better performance and scalability.

Our system is not very suitable for MPI clusters. MPI is network-bounded when processing large-scale dataset. Developers have to work on the parallelization and data distribution themselves. In addition, MPI lacks the fault tolerance feature, which is essential for large-scale long-time processing tasks.

7 CONCLUSIONS

We presented the design and implementation of two large-scale inter-app ICC risk analysis frameworks, namely MR-Droid and an improved and much larger-scale DIALDroid. Both frameworks enable scalable inter-app security analysis that have not been reported before. Our work confirms the existence of real-world collusive data leak and privilege escalation threats.

We highlight some of the security findings obtained by DIALDroid.

(1) We found no explicit-intent based collusive data leak or privilege escalation cases, but many cases in the implicit intent categories, e.g., 33,756 app pairs exhibiting both collusive data leak and privilege escalation behaviors. The results suggest that collusive data leak research needs to be focused more on implicit intents, rather than explicit intents.

(2) Although the total numbers of problematic ICCs and app pairs are extremely high, the number of sender apps initiating these ICCs is surprisingly small. E.g., the 5,638,772 inter-app ICCs exhibiting privilege escalation behaviors (without collusive data leaks) are originated from 134 sender apps. Similar properties are observed in other threat categories as well.

(3) We found that the majority of inter-app ICC links (> 99%) do not carry any sensitive data. This property implies that the typical workload of inter-app ICC analysis is much lower than the worst case workload (where all inter-app ICCs are sensitive).

(4) ACCESS_FINE_LOCATION and ACCESS_COARSE_LOCATION are by far the most leaked permissions via inter-app ICCs. These problematic inter-app ICCs pose threats to sensitive data and device security, regardless of whether they are due to malicious detection evasion, insecure development practices, or accidental matches.

(5) Our manual analysis reveals that accidental matches (via implicit intents) can occur when the intent filters in the manifest files of receiver apps are non-specific and loosely defined, while the code in the receiver apps have predicates restricting the types of data to be processed. Therefore, there exist false positives in our results. Accidental match is not a flaw of our ICC linking algorithm, as Android OS would produce the same linking. (Unmatched data types can crash the receiver apps at runtime.) How to improve the existing analysis approaches to incorporate program semantics is an open question.

(6) We found cases of same-developer privilege escalation app pairs, where the sender and receiver apps are from the same developer.

(7) When including the malware from Virus Share, more privilege escalation and collusive data leak cases are discovered (as expected).

Our MR-Droid work is under review at ACM CODASPY ‘17 Conference, and our DIALDroid work is under review at ACM ASIACCS ‘17 Conference.

8 REFERENCES

[1]. Mcafee labs threats report. http://www.mcafee.com/us/resources/reports/ rp-quarterly-threats-may-2016.pdf, 2016.

[2]. Arzt, Steven, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps.", *PLDI*, 2014.

[3]. Bagheri, Hamid, Alireza Sadeghi, Joshua Garcia, and Sam Malek. "Covert: Compositional analysis of android inter-app permission leakage." *IEEE Transactions on Software Engineering*, 2015

[4]. Bugiel, Sven, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. "Towards Taming Privilege-Escalation Attacks on Android.", *NDSS*, 2012.

[5]. Bugiel, Sven, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. "Xmandroid: A new android evolution to mitigate privilege escalation attacks.", Technical Report TR-2011-04 Technische Universität Darmstadt, 2011.

[6]. Bugiel, Sven, Stephen Heuser, and Ahmad-Reza Sadeghi. "Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies.", *22nd USENIX Security Symposium (USENIX Security 13)*, 2013.

[7]. Chin, Erika, Adrienne Porter Felt, Kate Greenwood, and David Wagner. "Analyzing inter-application communication in Android.", *9th international conference on Mobile systems, applications, and services*, pp. 239-252. ACM, 2011.

[8]. Davi, Lucas, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. "Privilege escalation attacks on android.", *International Conference on Information Security*, pp. 346-360. Springer Berlin Heidelberg, 2010.

[9]. Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters.", *Communications of the ACM* **51**, no. 1, 2008, 107-113.

[10]. Enck, William, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones." *ACM Transactions on Computer Systems (TOCS)*, **32**, no. 2, 2014, 5.

[11]. Fuchs, Adam P., Avik Chaudhuri, and Jeffrey S. Foster. "Scandroid: Automated security certification of android Applications." Technical report, Department of Computer Science, University of Maryland, College Park, 2009.

[12]. Gordon, Michael I., Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. "Information Flow Analysis of Android Applications in DroidSafe.", *NDSS*. 2015.

[13]. Hay, Roee, and Yair Amit. *Android browser cross-application scripting*. CVE-2011-2357, IBM Rational Application Security Research Group, 2011.

[14]. Hay, Roee, Omer Tripp, and Marco Pistoia. "Dynamic detection of inter-application communication vulnerabilities in Android.", *International Symposium on Software Testing and Analysis*, pp. 118-128. ACM, 2015.

[15]. He, Qing, Tianfeng Shang, Fuzhen Zhuang, and Zhongzhi Shi. “Parallel extreme learning machine for regression based on MapReduce.”, *Neurocomputing*, **102** (2013): 52-58.
[16]. Jing, Yiming, Gail-Joon Ahn, Adam Doupé, and Jeong Hyun Yi. “Checking Intent-based Communication in Android with Intent Space Analysis.”, *11th ACM on Asia Conference on Computer and Communications Security*, pp. 735-746. ACM, 2016.
[17]. Kantola, David, Erika Chin, Warren He, and David Wagner. “Reducing attack surfaces for intra-application communication in android.”, *The second ACM workshop on Security and privacy in smartphones and mobile devices*, pp. 69-80. ACM, 2012.
[18]. Klieber, William, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. “Android taint flow analysis for app sets.”, *The 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pp. 1-6. ACM, 2014.
[19]. Li, Feng, Beng Chin Ooi, M. Tamer Özsu, and Sai Wu. “Distributed data management using MapReduce.”, *ACM Computing Surveys (CSUR)* **46**, no. 3 (2014): 31.
[20]. Li, Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. “ApkCombiner: Combining multiple Android apps to support inter-app analysis.”, *IFIP International Information Security Conference*, pp. 513-527. Springer International Publishing, 2015.
[21]. Li, Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. “IccTA: Detecting inter-component privacy leaks in android apps.”, *International Conference on Software Engineering*, **1**, pp. 280-291. IEEE Press, 2015.
[22]. Liu, Fang, Xiaokui Shu, Danfeng Yao, and Ali R. Butt. “Privacy-preserving scanning of big content for sensitive data exposure with MapReduce.”, *ACM Conference on Data and Application Security and Privacy*, pp. 195-206. ACM, 2015.
[23]. Lu, Long, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. “Chex: statically vetting android apps for component hijacking vulnerabilities.”, *ACM conference on Computer and communications Security*, pp. 229-240. ACM, 2012.
[24]. Marforio, Claudio, Aurélien Francillon, Srdjan Capkun, Srdjan Capkun, and Srdjan Capkun. “Application collusion attack on the permission-based security model and its implications for modern smartphone systems.”, Zürich, Switzerland: Department of Computer Science, ETH Zurich, 2011.
[25]. Marforio, Claudio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. “Analysis of the communication between colluding applications on modern smartphones.”, *The 28th Annual Computer Security Applications Conference*, pp. 51-60. ACM, 2012.
[26]. Mulliner, Collin, Jon Oberheide, William Robertson, and Engin Kirda. “PatchDroid: scalable third-party security patches for Android devices.”, *The 29th Annual Computer Security Applications Conference*, pp. 259-268. ACM, 2013.
[27]. Mutchler, Patrick, Adam Doupé, John Mitchell, Chris Kruegel, and Giovanni Vigna. “A Large-Scale Study of Mobile Web App Security.”, *Mobile Security Technologies Workshop (MoST)*, 2015.
[28]. Octeau, Damien, Somesh Jha, Matthew Dering, Patrick McDaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. “Combining static analysis with probabilistic models to enable market-scale android inter-component analysis.”, *POPL*, 2016

[29]. Octeau, Damien, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. “Composite constant propagation: Application to android inter-component communication analysis.”, *International Conference on Software Engineering*, **1**, pp. 77-88. IEEE Press, 2015.

[30]. Octeau, Damien, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. “Effective inter-component communication mapping in android with Epicc: An essential step towards holistic security analysis.”, *USENIX Security Symposium (USENIX Security 13)*, pp. 543-558. 2013.

[31]. Rasthofer, Siegfried, Steven Arzt, Enrico Lovat, and Eric Bodden. “Droidforce: enforcing complex, data-centric, system-wide policies in android.”, *2014 Ninth International Conference on Availability, Reliability and Security (ARES)*, pp. 40-49. IEEE, 2014.

[32]. Ravitch, Tristan, E. Rogan Creswick, Aaron Tomb, Adam Foltzer, Trevor Elliott, and Ledah Casburn. “Multi-app security analysis with FUSE: Statically detecting android app collusion.”, *Program Protection and Reverse Engineering Workshop*, p. 4. ACM, 2014.

[33]. Ren, Chuangang, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. “Towards discovering and understanding task hijacking in android.”, *USENIX Security Symposium (USENIX Security 15)*, pp. 945-959. 2015.

[34]. Sbîrlea, Dragos, Michael G. Burke, Salvatore Guarnieri, Marco Pistoia, and Vivek Sarkar. “Automatic detection of inter-application permission leaks in Android applications.”, *IBM Journal of Research and Development*, **57**, no. 6 (2013): 10-1.

[35]. Schlegel, Roman, Kehuan Zhang, Xiao-yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. “Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones.”, *NDSS*, 2011.

[36]. Stuart, Jeff A., and John D. Owens. “Multi-GPU MapReduce on GPU clusters.”, *2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pp. 1068-1079. IEEE, 2011.

[37]. Terada, Takeshi, *Attacking Android browsers via intent scheme URLs*. Mitsui Bussan Secure Directions Inc., Technical report, 2014.

[38]. Wei, Fengguo, Sankardas Roy, and Xinming Ou. “Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps.”, *ACM SIGSAC Conference on Computer and Communications Security*, pp. 1329-1341. ACM, 2014.

[39]. Yang, Kun, Jianwei Zhuge, Yongke Wang, Lujue Zhou, and Haixin Duan. “IntentFuzzer: detecting capability leaks of android applications.”, *ACM symposium on Information, computer and communications security*, pp. 531-536. ACM, 2014.

[40]. Yang, Zhemin, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. “Appintent: Analyzing sensitive data transmission in android for privacy leakage detection.”, *ACM SIGSAC conference on Computer & communications security*, pp. 1043-1054. ACM, 2013.

[41]. Zaharia, Matei, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. “Spark: cluster computing with working sets.”, *HotCloud*, 2010.

[42]. Zhang, Mu, and Heng Yin. “AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications.”, *NDSS*, 2014.

[43]. Zhang, Xuyun, Laurence T. Yang, Chang Liu, and Jinjun Chen. “A scalable two-phase top-down specialization approach for data anonymization using mapreduce on cloud.”, *IEEE Transactions on Parallel and Distributed Systems* **25**, no. 2 (2014): 363-373.

[44]. Zhang, Yuan, Min Yang, Guofei Gu, and Hao Chen. “FineDroid: Enforcing Permissions with System-Wide Application Execution Context.”, *International Conference on Security and Privacy in Communication Systems*, pp. 3-22. Springer International Publishing, 2015.

[45]. Peng, Hao, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. “Using probabilistic generative models for ranking risks of android apps.” *ACM conference on Computer and communications security*, pp. 241-252., 2012.

[46]. Rasthofer, Siegfried, Steven Arzt, and Eric Bodden. “A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks.”, *NDSS*. 2014.

[47]. Octeau, Damien, Somesh Jha, and Patrick McDaniel. “Retargeting Android applications to Java bytecode.” *ACM SIGSOFT 20th international symposium on the foundations of software engineering*, p. 6., 2012.

[48]. Vallée-Rai, Raja, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. “Optimizing Java bytecode using the Soot framework: Is it feasible?” *International conference on compiler construction*, pp. 18-34. Springer Berlin Heidelberg, 2000.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

ICC	Inter-Component Communication
RAM	Random-Access Memory
URL	Uniform Resource Locator
SMS	Short Message Service
GPU	Graphics Processing Unit
CUDA	Compute Unified Device Architecture
MPI	Message Passing Interface